

GNU Unifont
17.0.01

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

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

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

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

1.3 Introduction

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

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

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

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

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

Buffer	Generic data structure for a linked list of buffer elements	??
Font	Data structure to hold information for one font	??
Glyph	Data structure to hold data for one bitmap glyph	??
NamePair	Data structure for a font ID number and name character string	??
Options	Data structure to hold options for OpenType font output	??
PARAMS	??
Table	Data structure for an OpenType table	??
TableRecord	Data structure for data associated with one OpenType table	??

Chapter 3

File Index

3.1 File List

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

src/ hangul.h	Define constants and function prototypes for using Hangul glyphs	??
src/ hex2otf.c	Hex2otf - Convert GNU Unifont .hex file to OpenType font	??
src/ hex2otf.h	Hex2otf.h - Header file for hex2otf.c	??
src/ johab2syllables.c	Create the Unicode Hangul Syllables block from component letters	??
src/ unibdf2hex.c	Unibdf2hex - Convert a BDF file into a unifont.hex file	??
src/ unibmp2hex.c	Unibmp2hex - Turn a .bmp or .wbmp glyph matrix into a GNU Unifont hex glyph set of 256 characters	??
src/ unibmpbump.c	Unibmpbump - Adjust a Microsoft bitmap (.bmp) file that was created by unihex2png but converted to .bmp	??
src/ unicoverage.c	Unicoverage - Show the coverage of Unicode plane scripts for a GNU Unifont hex glyph file	??
src/ unidup.c	Unidup - Check for duplicate code points in sorted unifont.hex file	??
src/ unifont-support.c	: Support functions for Unifont .hex files	??
src/ unifont1per.c	Unifont1per - Read a Unifont .hex file from standard input and produce one glyph per ".bmp" bitmap file as output	??
src/ unifontpic.c	Unifontpic - See the "Big Picture": the entire Unifont in one BMP bitmap	??
src/ unifontpic.h	Unifontpic.h - Header file for unifontpic.c	??
src/ unigen-hangul.c	Generate arbitrary hangul syllables	??

src/ unigencircles.c	Unigencircles - Superimpose dashed combining circles on combining glyphs	??
src/ unigenwidth.c	Unigenwidth - IEEE 1003.1-2008 setup to calculate wchar_t string widths	??
src/ unihangul-support.c	Functions for converting Hangul letters into syllables	??
src/ unihex2bmp.c	Unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points into a bitmap for editing	??
src/ unihexgen.c	Unihexgen - Generate a series of glyphs containing hexadecimal code points	??
src/ unihexpose.c	??
src/ unijohab2html.c	Display overlapped Hangul letter combinations in a grid	??
src/ unipagecount.c	Unipagecount - Count the number of glyphs defined in each page of 256 code points . . .	??

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/otf#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.](#)

```
00001 /**
00002  @file hangul.h
00003
00004  @brief Define constants and function prototypes for using Hangul glyphs.
00005
00006  @author Paul Hardy
00007
00008  @copyright Copyright © 2023 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 _HANGUL_H_
00028 #define _HANGUL_H_
00029
00030 #include <stdlib.h>
00031
00032
00033 #define MAXLINE 256 ///< Length of maximum file input line.
00034
```

```

00035 #define EXTENDED_HANGUL /* Use rare Hangul code points beyond U+1100 */
00036
00037 /* Definitions to move Hangul .hex file contents into the Private Use Area. */
00038 #define PUA_START 0xE000
00039 #define PUA_END 0xE8FF
00040 #define MAX_GLYPHS (PUA_END - PUA_START + 1) /* Maximum .hex file glyphs */
00041
00042 /*
00043     Unicode ranges for Hangul choseong, jungseong, and jongseong.
00044
00045     U+1100..U+11FF is the main range of modern and ancient Hangul jamo.
00046     U+A960..U+A97C is the range for extended Hangul choseong.
00047     U+D7B0..U+D7C6 is the range for extended Hangul jungseong.
00048     U+D7CB..U+D7FB is the range for extended Hangul jongseong.
00049 */
00050 #define CHO_UNICODE_START 0x1100 ///< Modern Hangul choseong start
00051 #define CHO_UNICODE_END 0x115E ///< Hangul Jamo choseong end
00052 #define CHO_EXTB_UNICODE_START 0xA960 ///< Hangul Extended-A choseong start
00053 #define CHO_EXTB_UNICODE_END 0xA97C ///< Hangul Extended-A choseong end
00054
00055 #define JUNG_UNICODE_START 0x1161 ///< Modern Hangul jungseong start
00056 #define JUNG_UNICODE_END 0x11A7 ///< Modern Hangul jungseong end
00057 #define JUNG_EXTB_UNICODE_START 0xD7B0 ///< Hangul Extended-B jungseong start
00058 #define JUNG_EXTB_UNICODE_END 0xD7C6 ///< Hangul Extended-B jungseong end
00059
00060 #define JONG_UNICODE_START 0x11A8 ///< Modern Hangul jongseong start
00061 #define JONG_UNICODE_END 0x11FF ///< Modern Hangul jongseong end
00062 #define JONG_EXTB_UNICODE_START 0xD7CB ///< Hangul Extended-B jongseong start
00063 #define JONG_EXTB_UNICODE_END 0xD7FB ///< Hangul Extended-B jongseong end
00064
00065
00066 /*
00067     Number of modern and ancient letters in hangul-base.hex file.
00068 */
00069 #define NCHO_MODERN 19 ///< 19 modern Hangul Jamo choseong
00070 #define NCHO_ANCIENT 76 ///< ancient Hangul Jamo choseong
00071 #define NCHO_EXTB 29 ///< Hangul Extended-A choseong
00072 #define NCHO_EXTB_RSRVD 3 ///< Reserved at end of Extended-A choseong
00073
00074 #define NJUNG_MODERN 21 ///< 21 modern Hangul Jamo jungseong
00075 #define NJUNG_ANCIENT 50 ///< ancient Hangul Jamo jungseong
00076 #define NJUNG_EXTB 23 ///< Hangul Extended-B jungseong
00077 #define NJUNG_EXTB_RSRVD 4 ///< Reserved at end of Extended-B jungseong
00078
00079 #define NJONG_MODERN 27 ///< 28 modern Hangul Jamo jongseong
00080 #define NJONG_ANCIENT 61 ///< ancient Hangul Jamo jongseong
00081 #define NJONG_EXTB 49 ///< Hangul Extended-B jongseong
00082 #define NJONG_EXTB_RSRVD 4 ///< Reserved at end of Extended-B jongseong
00083
00084
00085 /*
00086     Number of variations of each component in a Johab 6/3/1 arrangement.
00087 */
00088 #define CHO_VARIATIONS 6 ///< 6 choseong variations
00089 #define JUNG_VARIATIONS 3 ///< 3 jungseong variations
00090 #define JONG_VARIATIONS 1 ///< 1 jongseong variation
00091
00092 /*
00093     Starting positions in the hangul-base.hex file for each component.
00094 */
00095 ///< Location of first choseong (location 0x0000 is a blank glyph)
00096 #define CHO_HEX 0x0001
00097
00098 ///< Location of first ancient choseong
00099 #define CHO_ANCIENT_HEX (CHO_HEX + CHO_VARIATIONS * NCHO_MODERN)
0100
0101 ///< U+A960 Extended-A choseong
0102 #define CHO_EXTB_HEX (CHO_ANCIENT_HEX + CHO_VARIATIONS * NCHO_ANCIENT)
0103
0104 ///< U+A97F Extended-A last location in .hex file, including reserved Unicode code points at end
0105 #define CHO_LAST_HEX (CHO_EXTB_HEX + CHO_VARIATIONS * (NCHO_EXTB + NCHO_EXTB_RSRVD) - 1)
0106
0107 ///< Location of first jungseong (will be 0x2FB)
0108 #define JUNG_HEX (CHO_LAST_HEX + 1)
0109
0110 ///< Location of first ancient jungseong
0111 #define JUNG_ANCIENT_HEX (JUNG_HEX + JUNG_VARIATIONS * NJUNG_MODERN)
0112
0113 ///< U+D7B0 Extended-B jungseong
0114 #define JUNG_EXTB_HEX (JUNG_ANCIENT_HEX + JUNG_VARIATIONS * NJUNG_ANCIENT)
0115

```

```

00116 /// U+D7CA Extended-B last location in .hex file, including reserved Unicode code points at end
00117 #define JUNG_LAST_HEX (JUNG_EXTB_HEX + JUNG_VARIATIONS * (NJUNG_EXTB +
NJUNG_EXTB_RSRVD) - 1)
00118
00119 /// Location of first jongseong (will be 0x421)
00120 #define JONG_HEX (JUNG_LAST_HEX + 1)
00121
00122 /// Location of first ancient jongseong
00123 #define JONG_ANCIENT_HEX (JONG_HEX + JONG_VARIATIONS * NJONG_MODERN)
00124
00125 /// U+D7CB Extended-B jongseong
00126 #define JONG_EXTB_HEX (JONG_ANCIENT_HEX + JONG_VARIATIONS * NJONG_ANCIENT)
00127
00128 /// U+D7FF Extended-B last location in .hex file, including reserved Unicode code points at end
00129 #define JONG_LAST_HEX (JONG_EXTB_HEX + JONG_VARIATIONS * (NJONG_EXTB +
NJONG_EXTB_RSRVD) - 1)
00130
00131 /* Common modern and ancient Hangul Jamo range */
00132 #define JAMO_HEX 0x0500 ///< Start of U+1100..U+11FF glyphs
00133 #define JAMO_END 0x05FF ///< End of U+1100..U+11FF glyphs
00134
00135 /* Hangul Jamo Extended-A range */
00136 #define JAMO_EXT_A_HEX 0x0600 ///< Start of U+A960..U+A97F glyphs
00137 #define JAMO_EXT_A_END 0x061F ///< End of U+A960..U+A97F glyphs
00138
00139 /* Hangul Jamo Extended-B range */
00140 #define JAMO_EXTB_HEX 0x0620 ///< Start of U+D7B0..U+D7FF glyphs
00141 #define JAMO_EXTB_END 0x066F ///< End of U+D7B0..U+D7FF glyphs
00142
00143 /*
00144 These values allow enumeration of all modern and ancient letters.
00145
00146 If RARE_HANGUL is defined, include Hangul code points above U+11FF.
00147 */
00148 #ifdef EXTENDED_HANGUL
00149
00150 #define TOTAL_CHO (NCHO_MODERN + NCHO_ANCIENT + NCHO_EXT_A)
00151 #define TOTAL_JUNG (NJUNG_MODERN + NJUNG_ANCIENT + NJUNG_EXTB)
00152 #define TOTAL_JONG (NJONG_MODERN + NJONG_ANCIENT + NJONG_EXTB)
00153
00154 #else
00155
00156 #define TOTAL_CHO (NCHO_MODERN + NCHO_ANCIENT)
00157 #define TOTAL_JUNG (NJUNG_MODERN + NJUNG_ANCIENT)
00158 #define TOTAL_JONG (NJONG_MODERN + NJONG_ANCIENT)
00159
00160 #endif
00161
00162
00163 /*
00164 Function Prototypes.
00165 */
00166
00167 unsigned hangul_read_base8 (FILE *infp, unsigned char base[][32]);
00168 unsigned hangul_read_base16 (FILE *infp, unsigned base[][16]);
00169
00170 void hangul_decompose (unsigned codept,
00171 int *initial, int *medial, int *final);
00172 unsigned hangul_compose (int initial, int medial, int final);
00173
00174 void hangul_hex_indices (int choseong, int jungseong, int jongseong,
00175 int *cho_index, int *jung_index, int *jong_index);
00176 void hangul_variations (int choseong, int jungseong, int jongseong,
00177 int *cho_var, int *jung_var, int *jong_var);
00178 int is_wide_vowel (int vowel);
00179 int cho_variation (int choseong, int jungseong, int jongseong);
00180 int jung_variation (int choseong, int jungseong, int jongseong);
00181 int jong_variation (int choseong, int jungseong, int jongseong);
00182
00183 void hangul_syllable (int choseong, int jungseong, int jongseong,
00184 unsigned char hangul_base[][32], unsigned char *syllable);
00185 int glyph_overlap (unsigned *glyph1, unsigned *glyph2);
00186 void combine_glyphs (unsigned *glyph1, unsigned *glyph2,
00187 unsigned *combined_glyph);
00188 void one_jamo (unsigned glyph_table [MAX_GLYPHS][16],
00189 unsigned jamo, unsigned *jamo_glyph);
00190 void combined_jamo (unsigned glyph_table [MAX_GLYPHS][16],
00191 unsigned cho, unsigned jung, unsigned jong,
00192 unsigned *combined_glyph);
00193 void print_glyph_txt (FILE *fp, unsigned codept, unsigned *this_glyph);
00194 void print_glyph_hex (FILE *fp, unsigned codept, unsigned *this_glyph);

```

```
00195
00196
00197 #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"PRIxF32
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](#) (void)
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](#) (void)
- Print program version string on stdout.
- void [printHelp](#) (void)
- 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

Copyright © 2022 何志翔 (He Zhixiang)

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 definition 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"PRIFAST32
```

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 {
```

```

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

```

5.3.5 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](#).

```

00695 {
00696     Table *table = getBufferSlot (font->tables, sizeof (Table));
00697     table->tag = tagAsU32 (tag);
00698     table->content = content;
00699 }

```

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](#).

```

01162 {
01163     enum Direction {RIGHT, LEFT, DOWN, UP}; // order is significant
01164
01165     // respective coordinate deltas
01166     const pixels_t dx[] = {1, -1, 0, 0}, dy[] = {0, 0, -1, 1};
01167
01168     assert (byteCount % GLYPH_HEIGHT == 0);
01169     const uint_fast8_t bytesPerRow = byteCount / GLYPH_HEIGHT;
01170     const pixels_t glyphWidth = bytesPerRow * 8;
01171     assert (glyphWidth <= GLYPH_MAX_WIDTH);
01172
01173     #if GLYPH_MAX_WIDTH < 32
01174         typedef uint_fast32_t row_t;
01175     #elif GLYPH_MAX_WIDTH < 64
01176         typedef uint_fast64_t row_t;
01177     #else
01178         #error GLYPH_MAX_WIDTH is too large.
01179     #endif
01180
01181     row_t pixels[GLYPH_HEIGHT + 2] = {0};
01182     for (pixels_t row = GLYPH_HEIGHT; row > 0; row--)
01183         for (pixels_t b = 0; b < bytesPerRow; b++)
01184             pixels[row] = pixels[row] « 8 | *bitmap++;
01185     typedef row_t graph_t[GLYPH_HEIGHT + 1];
01186     graph_t vectors[4];
01187     const row_t *lower = pixels, *upper = pixels + 1;
01188     for (pixels_t row = 0; row <= GLYPH_HEIGHT; row++)
01189     {
01190         const row_t m = (fillSide == FILL_RIGHT) - 1;
01191         vectors[RIGHT][row] = (m ^ (*lower « 1)) & (~m ^ (*upper « 1));
01192         vectors[LEFT][row] = (m ^ (*upper )) & (~m ^ (*lower ));
01193         vectors[DOWN][row] = (m ^ (*lower )) & (~m ^ (*lower « 1));
01194         vectors[UP][row] = (m ^ (*upper « 1)) & (~m ^ (*upper ));
01195         lower++;
01196         upper++;
01197     }
01198     graph_t selection = {0};
01199     const row_t x0 = (row_t)1 « glyphWidth;
01200
01201     /// Get the value of a given bit that is in a given row.
01202     #define getRowBit(rows, x, y) ((rows)[(y)] & x0 » (x))
01203
01204     /// Invert the value of a given bit that is in a given row.
01205     #define flipRowBit(rows, x, y) ((rows)[(y)] ^ x0 » (x))
01206
01207     for (pixels_t y = GLYPH_HEIGHT; y >= 0; y--)
01208     {
01209         for (pixels_t x = 0; x <= glyphWidth; x++)
01210         {
01211             assert (!getRowBit (vectors[LEFT], x, y));
01212             assert (!getRowBit (vectors[UP], x, y));
01213             enum Direction initial;
01214
01215             if (getRowBit (vectors[RIGHT], x, y))
01216                 initial = RIGHT;
01217             else if (getRowBit (vectors[DOWN], x, y))
01218                 initial = DOWN;
01219             else
01220                 continue;
01221

```

```

01222     static_assert ((GLYPH_MAX_WIDTH + 1) * (GLYPH_HEIGHT + 1) * 2 <=
01223         U16MAX, "potential overflow");
01224
01225     uint_fast16_t lastPointCount = 0;
01226     for (bool converged = false;;)
01227     {
01228         uint_fast16_t pointCount = 0;
01229         enum Direction heading = initial;
01230         for (pixels_t tx = x, ty = y;;)
01231         {
01232             if (converged)
01233             {
01234                 storePixels (result, OP_POINT);
01235                 storePixels (result, tx);
01236                 storePixels (result, ty);
01237             }
01238             do
01239             {
01240                 if (converged)
01241                     flipRowBit (vectors[heading], tx, ty);
01242                 tx += dx[heading];
01243                 ty += dy[heading];
01244             } while (getRowBit (vectors[heading], tx, ty));
01245             if (tx == x && ty == y)
01246                 break;
01247             static_assert ((UP ^ DOWN) == 1 && (LEFT ^ RIGHT) == 1,
01248                 "wrong enums");
01249             heading = (heading & 2) ^ 2;
01250             heading |= !getRowBit (selection, tx, ty);
01251             heading ^= !getRowBit (vectors[heading], tx, ty);
01252             assert (getRowBit (vectors[heading], tx, ty));
01253             flipRowBit (selection, tx, ty);
01254             pointCount++;
01255         }
01256         if (converged)
01257             break;
01258         converged = pointCount == lastPointCount;
01259         lastPointCount = pointCount;
01260     }
01261
01262     storePixels (result, OP_CLOSE);
01263 }
01264 }
01265 #undef getRowBit
01266 #undef flipRowBit
01267 }

```

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](#).

```
01041 {
01042     const Glyph *const ga = a, *const gb = b;
01043     int gt = ga->codePoint > gb->codePoint;
01044     int lt = ga->codePoint < gb->codePoint;
01045     return gt - lt;
01046 }
```

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](#).

```
00768 {
00769     const struct TableRecord *const ra = a, *const rb = b;
00770     int gt = ra->tag > rb->tag;
00771     int lt = ra->tag < rb->tag;
00772     return gt - lt;
00773 }
```

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](#).

```

02317 {
02318     for (const char *p = str; *p; p++)
02319     {
02320         byte c = *p;
02321         if (c < 0x80)
02322         {
02323             cacheU16 (buf, c);
02324             continue;
02325         }
02326         int length = 1;
02327         byte mask = 0x40;
02328         for (; c & mask; mask >>= 1)
02329             length++;
02330         if (length == 1 || length > 4)
02331             fail ("Ill-formed UTF-8 sequence.");
02332         uint_fast32_t codePoint = c & (mask - 1);
02333         for (int i = 1; i < length; i++)
02334         {
02335             c = *p++;
02336             if ((c & 0xc0) != 0x80) // NUL checked here
02337                 fail ("Ill-formed UTF-8 sequence.");
02338             codePoint = (codePoint << 6) | (c & 0x3f);
02339         }
02340         const int lowerBits = length==2 ? 7 : length==3 ? 11 : 16;
02341         if (codePoint >> lowerBits == 0)
02342             fail ("Ill-formed UTF-8 sequence."); // sequence should be shorter
02343         if (codePoint >= 0xd800 && codePoint <= 0xdfff)
02344             fail ("Ill-formed UTF-8 sequence.");
02345         if (codePoint > 0x10ffff)
02346             fail ("Ill-formed UTF-8 sequence.");
02347         if (codePoint > 0xffff)
02348         {
02349             cacheU16 (buf, 0xd800 | (codePoint - 0x10000) >> 10);
02350             cacheU16 (buf, 0xdc00 | (codePoint & 0x3ff));
02351         }
02352         else
02353             cacheU16 (buf, codePoint);
02354     }
02355 }
```

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](#).

```

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

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](#).

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

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

5.3.5.13 cleanBuffers()

```
void cleanBuffers (
    void )
```

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](#).

```
00171 {
00172     for (size_t i = 0; i < bufferCount; i++)
00173         if (allBuffers[i].capacity)
00174             free (allBuffers[i].begin);
00175     free (allBuffers);
00176     bufferCount = 0;
00177 }
```

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](#).

```
00375 {
00376     assert (1 <= bytes && bytes <= 4);
00377     ensureBuffer (buf, bytes);
00378     switch (bytes)
00379     {
00380         case 4: *buf->next++ = value » 24 & 0xff; // fall through
00381         case 3: *buf->next++ = value » 16 & 0xff; // fall through
00382         case 2: *buf->next++ = value » 8 & 0xff; // fall through
00383         case 1: *buf->next++ = value & 0xff;
00384     }
00385 }
```


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](#).

```
00240 {
00241     if (buf->end - buf->next >= needed)
00242         return;
00243     ptrdiff_t occupied = buf->next - buf->begin;
00244     size_t required = occupied + needed;
00245     if (required < needed) // overflow
00246         fail ("Cannot allocate %zu + %zu bytes of memory.", occupied, needed);
00247     if (required > SIZE_MAX / 2)
00248         buf->capacity = required;
00249     else while (buf->capacity < required)
00250         buf->capacity *= 2;
00251     void *extended = realloc (buf->begin, buf->capacity);
00252     if (!extended)
00253         fail ("Failed to allocate %zu bytes of memory.", buf->capacity);
00254     buf->begin = extended;
00255     buf->next = buf->begin + occupied;
00256     buf->end = buf->begin + buf->capacity;
00257 }
```

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](#).

```
00114 {
00115     fputs ("ERROR: ", stderr);
00116     va_list args;
00117     va_start (args, reason);
00118     vfprintf (stderr, reason, args);
00119     va_end (args);
00120     putc ('\n', stderr);
00121     exit (EXIT_FAILURE);
00122 }
```

Here is the caller graph for this function:

5.3.5.17 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](#).

```
01729 {
01730     const Glyph *const glyphs = getBufferHead (font->glyphs);
01731     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
01732     size_t bitmapsSize = 0;
01733     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
01734         bitmapsSize += glyph->byteCount;
01735     Buffer *ebdt = newBuffer (4 + bitmapsSize);
01736     addTable (font, "EBDT", ebdt);
01737     cacheU16 (ebdt, 2); // majorVersion
01738     cacheU16 (ebdt, 0); // minorVersion
01739     uint_fast8_t byteCount = 0; // unequal to any glyph
01740     pixels_t pos = 0;
01741     bool combining = false;
01742     Buffer *rangeHeads = newBuffer (32);
01743     Buffer *offsets = newBuffer (64);
01744     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
01745     {
01746         if (glyph->byteCount != byteCount || glyph->pos != pos ||
01747             glyph->combining != combining)
01748         {
01749             storeU16 (rangeHeads, glyph - glyphs);
01750             storeU32 (offsets, countBufferedBytes (ebdt));
01751             byteCount = glyph->byteCount;
01752             pos = glyph->pos;
01753             combining = glyph->combining;
01754         }
01755         cacheBytes (ebdt, glyph->bitmap, byteCount);
01756     }
01757     const uint_least16_t *ranges = getBufferHead (rangeHeads);
01758     const uint_least16_t *rangesEnd = getBufferTail (rangeHeads);
01759     uint_fast32_t rangeCount = rangesEnd - ranges;
01760     storeU16 (rangeHeads, font->glyphCount);
01761     Buffer *eblc = newBuffer (4096);
01762     addTable (font, "EBLC", eblc);
01763     cacheU16 (eblc, 2); // majorVersion
01764     cacheU16 (eblc, 0); // minorVersion
01765     cacheU32 (eblc, 1); // numSizes
```

```

01766 { // bitmapSizes[0]
01767     cacheU32 (eblc, 56); // indexSubTableArrayOffset
01768     cacheU32 (eblc, (8 + 20) * rangeCount); // indexTablesSize
01769     cacheU32 (eblc, rangeCount); // numberOfIndexSubTables
01770     cacheU32 (eblc, 0); // colorRef
01771     { // hori
01772         cacheU8 (eblc, ASCENDER); // ascender
01773         cacheU8 (eblc, -DESCENDER); // descender
01774         cacheU8 (eblc, font->maxWidth); // widthMax
01775         cacheU8 (eblc, 1); // caretSlopeNumerator
01776         cacheU8 (eblc, 0); // caretSlopeDenominator
01777         cacheU8 (eblc, 0); // caretOffset
01778         cacheU8 (eblc, 0); // minOriginSB
01779         cacheU8 (eblc, 0); // minAdvanceSB
01780         cacheU8 (eblc, ASCENDER); // maxBeforeBL
01781         cacheU8 (eblc, -DESCENDER); // minAfterBL
01782         cacheU8 (eblc, 0); // pad1
01783         cacheU8 (eblc, 0); // pad2
01784     }
01785     { // vert
01786         cacheU8 (eblc, ASCENDER); // ascender
01787         cacheU8 (eblc, -DESCENDER); // descender
01788         cacheU8 (eblc, font->maxWidth); // widthMax
01789         cacheU8 (eblc, 1); // caretSlopeNumerator
01790         cacheU8 (eblc, 0); // caretSlopeDenominator
01791         cacheU8 (eblc, 0); // caretOffset
01792         cacheU8 (eblc, 0); // minOriginSB
01793         cacheU8 (eblc, 0); // minAdvanceSB
01794         cacheU8 (eblc, ASCENDER); // maxBeforeBL
01795         cacheU8 (eblc, -DESCENDER); // minAfterBL
01796         cacheU8 (eblc, 0); // pad1
01797         cacheU8 (eblc, 0); // pad2
01798     }
01799     cacheU16 (eblc, 0); // startGlyphIndex
01800     cacheU16 (eblc, font->glyphCount - 1); // endGlyphIndex
01801     cacheU8 (eblc, 16); // ppemX
01802     cacheU8 (eblc, 16); // ppemY
01803     cacheU8 (eblc, 1); // bitDepth
01804     cacheU8 (eblc, 1); // flags = Horizontal
01805 }
01806 { // IndexSubTableArray
01807     uint_fast32_t offset = rangeCount * 8;
01808     for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
01809     {
01810         cacheU16 (eblc, *p); // firstGlyphIndex
01811         cacheU16 (eblc, p[1] - 1); // lastGlyphIndex
01812         cacheU32 (eblc, offset); // additionalOffsetToIndexSubtable
01813         offset += 20;
01814     }
01815 }
01816 { // IndexSubTables
01817     const uint_least32_t *offset = getBufferHead (offsets);
01818     for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
01819     {
01820         const Glyph *glyph = &glyphs[*p];
01821         cacheU16 (eblc, 2); // indexFormat
01822         cacheU16 (eblc, 5); // imageFormat
01823         cacheU32 (eblc, *offset++); // imageDataOffset
01824         cacheU32 (eblc, glyph->byteCount); // imageSize
01825         { // bigMetrics
01826             cacheU8 (eblc, GLYPH_HEIGHT); // height
01827             const uint_fast8_t width = PW (glyph->byteCount);
01828             cacheU8 (eblc, width); // width
01829             cacheU8 (eblc, glyph->pos); // horiBearingX
01830             cacheU8 (eblc, ASCENDER); // horiBearingY
01831             cacheU8 (eblc, glyph->combining ? 0 : width); // horiAdvance
01832             cacheU8 (eblc, 0); // vertBearingX
01833             cacheU8 (eblc, 0); // vertBearingY
01834             cacheU8 (eblc, GLYPH_HEIGHT); // vertAdvance
01835         }
01836     }
01837 }
01838 freeBuffer (rangeHeads);
01839 freeBuffer (offsets);
01840 }

```

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](#).

```
01698 {
01699     Buffer *glyph = newBuffer (12);
01700     addTable (font, "glyf", glyph);
01701     // Empty table is not allowed, but an empty outline for glyph 0 suffices.
01702     cacheU16 (glyph, 0); // numberOfContours
01703     cacheU16 (glyph, FU (0)); // xMin
01704     cacheU16 (glyph, FU (0)); // yMin
01705     cacheU16 (glyph, FU (0)); // xMax
01706     cacheU16 (glyph, FU (0)); // yMax
01707     cacheU16 (glyph, 0); // instructionLength
01708     Buffer *loca = newBuffer (2 * (font->glyphCount + 1));
01709     addTable (font, "loca", loca);
01710     cacheU16 (loca, 0); // offsets[0]
01711     assert (countBufferedBytes (glyph) % 2 == 0);
01712     for (uint_fast32_t i = 1; i <= font->glyphCount; i++)
01713         cacheU16 (loca, countBufferedBytes (glyph) / 2); // offsets[i]
01714 }
```

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](#).

```
01330 {
01331     // HACK: For convenience, CFF data structures are hard coded.
01332     assert (0 < version && version <= 2);
01333     Buffer *cff = newBuffer (65536);
01334     addTable (font, version == 1 ? "CFF " : "CFF2", cff);
01335 }
```

```

01336  /// Use fixed width integer for variables to simplify offset calculation.
01337  #define cacheCFF32(buf, x) (cacheU8 ((buf), 29), cacheU32 ((buf), (x)))
01338
01339  /// In Unifont, 16px glyphs are more common. This is used by CFF1 only.
01340  const pixels_t defaultWidth = 16, nominalWidth = 8;
01341  if (version == 1)
01342  {
01343      Buffer *strings = prepareStringIndex (names);
01344      size_t stringsSize = countBufferedBytes (strings);
01345      const char *cffName = names[6];
01346      assert (cffName);
01347      size_t nameLength = strlen (cffName);
01348      size_t namesSize = nameLength + 5;
01349      /// These sizes must be updated together with the data below.
01350      size_t offsets[] = {4, namesSize, 45, stringsSize, 2, 5, 8, 32, 4, 0};
01351      prepareOffsets (offsets);
01352      { /// Header
01353          cacheU8 (cff, 1); /// major
01354          cacheU8 (cff, 0); /// minor
01355          cacheU8 (cff, 4); /// hdrSize
01356          cacheU8 (cff, 1); /// offSize
01357      }
01358      assert (countBufferedBytes (cff) == offsets[0]);
01359      { /// Name INDEX (should not be used by OpenType readers)
01360          cacheU16 (cff, 1); /// count
01361          cacheU8 (cff, 1); /// offSize
01362          cacheU8 (cff, 1); /// offset[0]
01363          if (nameLength + 1 > 255) /// must be too long; spec limit is 63
01364              fail ("PostScript name is too long.");
01365          cacheU8 (cff, nameLength + 1); /// offset[1]
01366          cacheBytes (cff, cffName, nameLength);
01367      }
01368      assert (countBufferedBytes (cff) == offsets[1]);
01369      { /// Top DICT INDEX
01370          cacheU16 (cff, 1); /// count
01371          cacheU8 (cff, 1); /// offSize
01372          cacheU8 (cff, 1); /// offset[0]
01373          cacheU8 (cff, 41); /// offset[1]
01374          cacheCFFOperand (cff, 391); /// "Adobe"
01375          cacheCFFOperand (cff, 392); /// "Identity"
01376          cacheCFFOperand (cff, 0);
01377          cacheBytes (cff, (byte[]) {12, 30}, 2); /// ROS
01378          cacheCFF32 (cff, font->glyphCount);
01379          cacheBytes (cff, (byte[]) {12, 34}, 2); /// CIDCount
01380          cacheCFF32 (cff, offsets[6]);
01381          cacheBytes (cff, (byte[]) {12, 36}, 2); /// FDArray
01382          cacheCFF32 (cff, offsets[5]);
01383          cacheBytes (cff, (byte[]) {12, 37}, 2); /// FDSelect
01384          cacheCFF32 (cff, offsets[4]);
01385          cacheU8 (cff, 15); /// charSet
01386          cacheCFF32 (cff, offsets[8]);
01387          cacheU8 (cff, 17); /// CharStrings
01388      }
01389      assert (countBufferedBytes (cff) == offsets[2]);
01390      { /// String INDEX
01391          cacheBuffer (cff, strings);
01392          freeBuffer (strings);
01393      }
01394      assert (countBufferedBytes (cff) == offsets[3]);
01395      cacheU16 (cff, 0); /// Global Subr INDEX
01396      assert (countBufferedBytes (cff) == offsets[4]);
01397      { /// Charsets
01398          cacheU8 (cff, 2); /// format
01399          { /// Range2[0]
01400              cacheU16 (cff, 1); /// first
01401              cacheU16 (cff, font->glyphCount - 2); /// nLeft
01402          }
01403      }
01404      assert (countBufferedBytes (cff) == offsets[5]);
01405      { /// FDSelect
01406          cacheU8 (cff, 3); /// format
01407          cacheU16 (cff, 1); /// nRanges
01408          cacheU16 (cff, 0); /// first
01409          cacheU8 (cff, 0); /// fd
01410          cacheU16 (cff, font->glyphCount); /// sentinel
01411      }
01412      assert (countBufferedBytes (cff) == offsets[6]);
01413      { /// FDArray
01414          cacheU16 (cff, 1); /// count
01415          cacheU8 (cff, 1); /// offSize
01416          cacheU8 (cff, 1); /// offset[0]

```

```

01417     cacheU8 (cff, 28); // offset[1]
01418     cacheCFFOperand (cff, 393);
01419     cacheBytes (cff, (byte[]){12, 38}, 2); // FontName
01420     // Windows requires FontMatrix in Font DICT.
01421     const byte unit[] = {0x1e,0x15,0x62,0x5c,0x6f}; // 1/64 (0.015625)
01422     cacheBytes (cff, unit, sizeof unit);
01423     cacheCFFOperand (cff, 0);
01424     cacheCFFOperand (cff, 0);
01425     cacheBytes (cff, unit, sizeof unit);
01426     cacheCFFOperand (cff, 0);
01427     cacheCFFOperand (cff, 0);
01428     cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
01429     cacheCFFOperand (cff, offsets[8] - offsets[7]); // size
01430     cacheCFF32 (cff, offsets[7]); // offset
01431     cacheU8 (cff, 18); // Private
01432 }
01433 assert (countBufferedBytes (cff) == offsets[7]);
01434 { // Private
01435     cacheCFFOperand (cff, FU (defaultWidth));
01436     cacheU8 (cff, 20); // defaultWidthX
01437     cacheCFFOperand (cff, FU (nominalWidth));
01438     cacheU8 (cff, 21); // nominalWidthX
01439 }
01440 assert (countBufferedBytes (cff) == offsets[8]);
01441 }
01442 else
01443 {
01444     assert (version == 2);
01445     // These sizes must be updated together with the data below.
01446     size_t offsets[] = {5, 21, 4, 10, 0};
01447     prepareOffsets (offsets);
01448     { // Header
01449         cacheU8 (cff, 2); // majorVersion
01450         cacheU8 (cff, 0); // minorVersion
01451         cacheU8 (cff, 5); // headerSize
01452         cacheU16 (cff, offsets[1] - offsets[0]); // topDictLength
01453     }
01454     assert (countBufferedBytes (cff) == offsets[0]);
01455     { // Top DICT
01456         const byte unit[] = {0x1e,0x15,0x62,0x5c,0x6f}; // 1/64 (0.015625)
01457         cacheBytes (cff, unit, sizeof unit);
01458         cacheCFFOperand (cff, 0);
01459         cacheCFFOperand (cff, 0);
01460         cacheBytes (cff, unit, sizeof unit);
01461         cacheCFFOperand (cff, 0);
01462         cacheCFFOperand (cff, 0);
01463         cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
01464         cacheCFFOperand (cff, offsets[2]);
01465         cacheBytes (cff, (byte[]){12, 36}, 2); // FDArray
01466         cacheCFFOperand (cff, offsets[3]);
01467         cacheU8 (cff, 17); // CharStrings
01468     }
01469     assert (countBufferedBytes (cff) == offsets[1]);
01470     cacheU32 (cff, 0); // Global Subr INDEX
01471     assert (countBufferedBytes (cff) == offsets[2]);
01472     { // Font DICT INDEX
01473         cacheU32 (cff, 1); // count
01474         cacheU8 (cff, 1); // offSize
01475         cacheU8 (cff, 1); // offset[0]
01476         cacheU8 (cff, 4); // offset[1]
01477         cacheCFFOperand (cff, 0);
01478         cacheCFFOperand (cff, 0);
01479         cacheU8 (cff, 18); // Private
01480     }
01481     assert (countBufferedBytes (cff) == offsets[3]);
01482 }
01483 { // CharStrings INDEX
01484     Buffer *offsets = newBuffer (4096);
01485     Buffer *charstrings = newBuffer (4096);
01486     Buffer *outline = newBuffer (1024);
01487     const Glyph *glyph = getBufferHead (font->glyphs);
01488     const Glyph *const endGlyph = glyph + font->glyphCount;
01489     for (; glyph < endGlyph; glyph++)
01490     {
01491         // CFF offsets start at 1
01492         storeU32 (offsets, countBufferedBytes (charstrings) + 1);
01493
01494         pixels_t rx = -glyph->pos;
01495         pixels_t ry = DESCENDER;
01496         resetBuffer (outline);
01497         buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_LEFT);

```

```

01498     enum CFFOp {rmoveto=21, hmoveto=22, vmoveto=4, hlineto=6,
01499               vlineto=7, endchar=14};
01500     enum CFFOp pendingOp = 0;
01501     const int STACK_LIMIT = version == 1 ? 48 : 513;
01502     int stackSize = 0;
01503     bool isDrawing = false;
01504     pixels_t width = glyph->combining ? 0 : PW (glyph->byteCount);
01505     if (version == 1 && width != defaultWidth)
01506     {
01507         cacheCFFOperand (charstrings, FU (width - nominalWidth));
01508         stackSize++;
01509     }
01510     for (const pixels_t *p = getBufferHead (outline),
01511          *const end = getBufferTail (outline); p < end;)
01512     {
01513         int s = 0;
01514         const enum ContourOp op = *p++;
01515         if (op == OP_POINT)
01516         {
01517             const pixels_t x = *p++, y = *p++;
01518             if (x != rx)
01519             {
01520                 cacheCFFOperand (charstrings, FU (x - rx));
01521                 rx = x;
01522                 stackSize++;
01523                 s |= 1;
01524             }
01525             if (y != ry)
01526             {
01527                 cacheCFFOperand (charstrings, FU (y - ry));
01528                 ry = y;
01529                 stackSize++;
01530                 s |= 2;
01531             }
01532             assert (!(isDrawing && s == 3));
01533         }
01534         if (s)
01535         {
01536             if (!isDrawing)
01537             {
01538                 const enum CFFOp moves[] = {0, hmoveto, vmoveto,
01539                                             rmoveto};
01540                 cacheU8 (charstrings, moves[s]);
01541                 stackSize = 0;
01542             }
01543             else if (!pendingOp)
01544                 pendingOp = (enum CFFOp[]){0, hlineto, vlineto}[s];
01545         }
01546         else if (!isDrawing)
01547         {
01548             // only when the first point happens to be (0, 0)
01549             cacheCFFOperand (charstrings, FU (0));
01550             cacheU8 (charstrings, hmoveto);
01551             stackSize = 0;
01552         }
01553         if (op == OP_CLOSE || stackSize >= STACK_LIMIT)
01554         {
01555             assert (stackSize <= STACK_LIMIT);
01556             cacheU8 (charstrings, pendingOp);
01557             pendingOp = 0;
01558             stackSize = 0;
01559         }
01560         isDrawing = op != OP_CLOSE;
01561     }
01562     if (version == 1)
01563         cacheU8 (charstrings, endchar);
01564 }
01565 size_t lastOffset = countBufferedBytes (charstrings) + 1;
01566 #if SIZE_MAX > U32MAX
01567     if (lastOffset > U32MAX)
01568         fail ("CFF data exceeded size limit.");
01569 #endif
01570 storeU32 (offsets, lastOffset);
01571 int offsetSize = 1 + (lastOffset > 0xff)
01572               + (lastOffset > 0xffff)
01573               + (lastOffset > 0xfffff);
01574 // count (must match 'numGlyphs' in 'maxp' table)
01575 cacheU (cff, font->glyphCount, version * 2);
01576 cacheU8 (cff, offsetSize); // offSize
01577 const uint_least32_t *p = getBufferHead (offsets);
01578 const uint_least32_t *const end = getBufferTail (offsets);

```

```

01579     for (; p < end; p++)
01580         cacheU (cff, *p, offsetSize); // offsets
01581     cacheBuffer (cff, charstrings); // data
01582     freeBuffer (offsets);
01583     freeBuffer (charstrings);
01584     freeBuffer (outline);
01585 }
01586 #undef cacheCFF32
01587 }

```

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](#).

```

02110 {
02111     Glyph *const glyphs = getBufferHead (font->glyphs);
02112     Buffer *rangeHeads = newBuffer (16);
02113     uint_fast32_t rangeCount = 0;
02114     uint_fast32_t bmpRangeCount = 1; // 1 for the last 0xffff-0xffff range
02115     glyphs[0].codePoint = glyphs[1].codePoint; // to start a range at glyph 1
02116     for (uint_fast16_t i = 1; i < font->glyphCount; i++)
02117     {
02118         if (glyphs[i].codePoint != glyphs[i - 1].codePoint + 1)
02119         {
02120             storeU16 (rangeHeads, i);
02121             rangeCount++;
02122             bmpRangeCount += glyphs[i].codePoint < 0xffff;
02123         }
02124     }
02125     Buffer *cmap = newBuffer (256);
02126     addTable (font, "cmap", cmap);
02127     // Format 4 table is always generated for compatibility.
02128     bool hasFormat12 = glyphs[font->glyphCount - 1].codePoint > 0xffff;
02129     cacheU16 (cmap, 0); // version
02130     cacheU16 (cmap, 1 + hasFormat12); // numTables
02131     { // encodingRecords[0]
02132         cacheU16 (cmap, 3); // platformID
02133         cacheU16 (cmap, 1); // encodingID
02134         cacheU32 (cmap, 12 + 8 * hasFormat12); // subtableOffset
02135     }
02136     if (hasFormat12) // encodingRecords[1]
02137     {
02138         cacheU16 (cmap, 3); // platformID
02139         cacheU16 (cmap, 10); // encodingID
02140         cacheU32 (cmap, 36 + 8 * bmpRangeCount); // subtableOffset
02141     }
02142     const uint_least16_t *ranges = getBufferHead (rangeHeads);
02143     const uint_least16_t *const rangesEnd = getBufferTail (rangeHeads);
02144     storeU16 (rangeHeads, font->glyphCount);
02145     { // format 4 table
02146         cacheU16 (cmap, 4); // format
02147         cacheU16 (cmap, 16 + 8 * bmpRangeCount); // length
02148         cacheU16 (cmap, 0); // language
02149         if (bmpRangeCount * 2 > U16MAX)
02150             fail ("Too many ranges in 'cmap' table.");
02151     }
02152 }

```



```

02151     cacheU16 (cmap, bmpRangeCount * 2); // segCountX2
02152     uint_fast16_t searchRange = 1, entrySelector = -1;
02153     while (searchRange <= bmpRangeCount)
02154     {
02155         searchRange <<= 1;
02156         entrySelector++;
02157     }
02158     cacheU16 (cmap, searchRange); // searchRange
02159     cacheU16 (cmap, entrySelector); // entrySelector
02160     cacheU16 (cmap, bmpRangeCount * 2 - searchRange); // rangeShift
02161     { // endCode[]
02162         const uint_least16_t *p = ranges;
02163         for (p++; p < rangesEnd && glyphs[*p].codePoint < 0xffff; p++)
02164             cacheU16 (cmap, glyphs[*p - 1].codePoint);
02165         uint_fast32_t cp = glyphs[*p - 1].codePoint;
02166         if (cp > 0xfffe)
02167             cp = 0xffff;
02168         cacheU16 (cmap, cp);
02169         cacheU16 (cmap, 0xffff);
02170     }
02171     cacheU16 (cmap, 0); // reservedPad
02172     { // startCode[]
02173         for (uint_fast32_t i = 0; i < bmpRangeCount - 1; i++)
02174             cacheU16 (cmap, glyphs[ranges[i]].codePoint);
02175         cacheU16 (cmap, 0xffff);
02176     }
02177     { // idDelta[]
02178         const uint_least16_t *p = ranges;
02179         for (; p < rangesEnd && glyphs[*p].codePoint < 0xffff; p++)
02180             cacheU16 (cmap, *p - glyphs[*p].codePoint);
02181         uint_fast16_t delta = 1;
02182         if (p < rangesEnd && *p == 0xffff)
02183             delta = *p - glyphs[*p].codePoint;
02184         cacheU16 (cmap, delta);
02185     }
02186     { // idRangeOffsets[]
02187         for (uint_least16_t i = 0; i < bmpRangeCount; i++)
02188             cacheU16 (cmap, 0);
02189     }
02190 }
02191 if (hasFormat12) // format 12 table
02192 {
02193     cacheU16 (cmap, 12); // format
02194     cacheU16 (cmap, 0); // reserved
02195     cacheU32 (cmap, 16 + 12 * rangeCount); // length
02196     cacheU32 (cmap, 0); // language
02197     cacheU32 (cmap, rangeCount); // numGroups
02198
02199     // groups[]
02200     for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
02201     {
02202         cacheU32 (cmap, glyphs[*p].codePoint); // startCharCode
02203         cacheU32 (cmap, glyphs[p[1] - 1].codePoint); // endCharCode
02204         cacheU32 (cmap, *p); // startGlyphID
02205     }
02206 }
02207 freeBuffer (rangeHeads);
02208 }

```

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](#).

```

02242 {
02243     Buffer *gpos = newBuffer (16);
02244     addTable (font, "GPOS", gpos);
02245     cacheU16 (gpos, 1); // majorVersion
02246     cacheU16 (gpos, 0); // minorVersion
02247     cacheU16 (gpos, 10); // scriptListOffset
02248     cacheU16 (gpos, 12); // featureListOffset
02249     cacheU16 (gpos, 14); // lookupListOffset
02250     { // ScriptList table
02251         cacheU16 (gpos, 0); // scriptCount
02252     }
02253     { // Feature List table
02254         cacheU16 (gpos, 0); // featureCount
02255     }
02256     { // Lookup List Table
02257         cacheU16 (gpos, 0); // lookupCount
02258     }
02259 }
```

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](#).

```

02270 {
02271     Buffer *gsub = newBuffer (38);
02272     addTable (font, "GSUB", gsub);
02273     cacheU16 (gsub, 1); // majorVersion
02274     cacheU16 (gsub, 0); // minorVersion
02275     cacheU16 (gsub, 10); // scriptListOffset
02276     cacheU16 (gsub, 34); // featureListOffset
02277     cacheU16 (gsub, 36); // lookupListOffset
02278     { // ScriptList table
02279         cacheU16 (gsub, 2); // scriptCount
02280         { // scriptRecords[0]
02281             cacheBytes (gsub, "DFLT", 4); // scriptTag
02282             cacheU16 (gsub, 14); // scriptOffset
02283         }
02284         { // scriptRecords[1]
02285             cacheBytes (gsub, "thai", 4); // scriptTag
02286             cacheU16 (gsub, 14); // scriptOffset
02287         }
02288     } // Script table
02289     cacheU16 (gsub, 4); // defaultLangSysOffset
02290     cacheU16 (gsub, 0); // langSysCount
02291     { // Default Language System table
02292         cacheU16 (gsub, 0); // lookupOrderOffset
```

```

02293         cacheU16 (gsub, 0); // requiredFeatureIndex
02294         cacheU16 (gsub, 0); // featureIndexCount
02295     }
02296 }
02297 }
02298 { // Feature List table
02299     cacheU16 (gsub, 0); // featureCount
02300 }
02301 { // Lookup List Table
02302     cacheU16 (gsub, 0); // lookupCount
02303 }
02304 }

```

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](#).

```

01854 {
01855     Buffer *head = newBuffer (56);
01856     addTable (font, "head", head);
01857     cacheU16 (head, 1); // majorVersion
01858     cacheU16 (head, 0); // minorVersion
01859     cacheZeros (head, 4); // fontRevision (unused)
01860     // The 'checksumAdjustment' field is a checksum of the entire file.
01861     // It is later calculated and written directly in the 'writeFont' function.
01862     cacheU32 (head, 0); // checksumAdjustment (placeholder)
01863     cacheU32 (head, 0x5f0f3cf5); // magicNumber
01864     const uint_fast16_t flags =
01865         + B1 ( 0) // baseline at y=0
01866         + B1 ( 1) // LSB at x=0 (doubtful; probably should be LSB=xMin)
01867         + B0 ( 2) // instructions may depend on point size
01868         + B0 ( 3) // force internal ppem to integers
01869         + B0 ( 4) // instructions may alter advance width
01870         + B0 ( 5) // not used in OpenType
01871         + B0 ( 6) // not used in OpenType
01872         + B0 ( 7) // not used in OpenType
01873         + B0 ( 8) // not used in OpenType
01874         + B0 ( 9) // not used in OpenType
01875         + B0 (10) // not used in OpenType
01876         + B0 (11) // font transformed
01877         + B0 (12) // font converted
01878         + B0 (13) // font optimized for ClearType
01879         + B0 (14) // last resort font
01880         + B0 (15) // reserved
01881     ;
01882     cacheU16 (head, flags); // flags
01883     cacheU16 (head, FUPEM); // unitsPerEm
01884     cacheZeros (head, 8); // created (unused)
01885     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); // numberOfMetrics
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](#).

```
02088 {
02089     Buffer *hmtx = newBuffer (4 * font->glyphCount);
02090     addTable (font, "hmtx", hmtx);
02091     const Glyph *const glyphs = getBufferHead (font->glyphs);
02092     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
02093     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
02094     {
02095         int\_fast16\_t aw = glyph->combining ? 0 : PW (glyph->byteCount);
02096         cacheU16 (hmtx, FU (aw)); // advanceWidth
02097         cacheU16 (hmtx, FU (glyph->lsb)); // lsb
02098     }
02099 }
```

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](#).

```
01956 {
01957     Buffer *maxp = newBuffer (32);
01958     addTable (font, "maxp", maxp);
01959     cacheU32 (maxp, isCFF ? 0x00005000 : 0x00010000); // version
01960     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);

```

```
02398 }
```

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](#).

```
01987 {
01988     Buffer *os2 = newBuffer (100);
01989     addTable (font, "OS/2", os2);
01990     cacheU16 (os2, 5); // version
01991     // HACK: Average glyph width is not actually calculated.
01992     cacheU16 (os2, FU (font->maxWidth)); // xAvgCharWidth
01993     cacheU16 (os2, 400); // usWeightClass = Normal
01994     cacheU16 (os2, 5); // usWidthClass = Medium
01995     const uint\_fast16\_t typeFlags =
01996         + B0 (0) // reserved
01997         // usage permissions, one of:
01998         // Default: Installable embedding
01999         + B0 (1) // Restricted License embedding
02000         + B0 (2) // Preview & Print embedding
02001         + B0 (3) // Editable embedding
02002         // 4-7 reserved
02003         + B0 (8) // no subsetting
02004         + B0 (9) // bitmap embedding only
02005         // 10-15 reserved
02006     ;
02007     cacheU16 (os2, typeFlags); // fsType
02008     cacheU16 (os2, FU (5)); // ySubscriptXSize
02009     cacheU16 (os2, FU (7)); // ySubscriptYSize
02010     cacheU16 (os2, FU (0)); // ySubscriptXOffset
02011     cacheU16 (os2, FU (1)); // ySubscriptYOffset
02012     cacheU16 (os2, FU (5)); // ySuperscriptXSize
02013     cacheU16 (os2, FU (7)); // ySuperscriptYSize
02014     cacheU16 (os2, FU (0)); // ySuperscriptXOffset
02015     cacheU16 (os2, FU (4)); // ySuperscriptYOffset
02016     cacheU16 (os2, FU (1)); // yStrikeoutSize
02017     cacheU16 (os2, FU (5)); // yStrikeoutPosition
02018     cacheU16 (os2, 0x080a); // sFamilyClass = Sans Serif, Matrix
02019     const byte panose[] =
02020     {
02021         2, // Family Kind = Latin Text
02022         11, // Serif Style = Normal Sans
02023         4, // Weight = Thin
02024         // Windows would render all glyphs to the same width,
02025         // if 'Proportion' is set to 'Monospaced' (as Unifont should be).
02026         // 'Condensed' is the best alternative according to metrics.
02027         6, // Proportion = Condensed
02028         2, // Contrast = None
02029         2, // Stroke = No Variation
02030         2, // Arm Style = Straight Arms
02031         8, // Letterform = Normal/Square
02032         2, // Midline = Standard/Trimmed
02033         4, // X-height = Constant/Large
02034     };
02035     cacheBytes (os2, panose, sizeof panose); // panose
```

```

02036 // HACK: All defined Unicode ranges are marked functional for convenience.
02037 cacheU32 (os2, 0xffffffff); // ulUnicodeRange1
02038 cacheU32 (os2, 0xffffffff); // ulUnicodeRange2
02039 cacheU32 (os2, 0xffffffff); // ulUnicodeRange3
02040 cacheU32 (os2, 0x0effffff); // ulUnicodeRange4
02041 cacheBytes (os2, "GNU ", 4); // achVendID
02042 // fsSelection (must agree with 'macStyle' in 'head' table)
02043 const uint_fast16_t selection =
02044     + B0 (0) // italic
02045     + B0 (1) // underscored
02046     + B0 (2) // negative
02047     + B0 (3) // outlined
02048     + B0 (4) // strikeout
02049     + B0 (5) // bold
02050     + B1 (6) // regular
02051     + B1 (7) // use sTypo* metrics in this table
02052     + B1 (8) // font name conforms to WWS model
02053     + B0 (9) // oblique
02054     // 10-15 reserved
02055 ;
02056 cacheU16 (os2, selection);
02057 const Glyph *glyphs = getBufferHead (font->glyphs);
02058 uint_fast32_t first = glyphs[1].codePoint;
02059 uint_fast32_t last = glyphs[font->glyphCount - 1].codePoint;
02060 cacheU16 (os2, first < U16MAX ? first : U16MAX); // usFirstCharIndex
02061 cacheU16 (os2, last < U16MAX ? last : U16MAX); // usLastCharIndex
02062 cacheU16 (os2, FU (ASCENDER)); // sTypoAscender
02063 cacheU16 (os2, FU (-DESCENDER)); // sTypoDescender
02064 cacheU16 (os2, FU (0)); // sTypoLineGap
02065 cacheU16 (os2, FU (ASCENDER)); // usWinAscent
02066 cacheU16 (os2, FU (DESCENDER)); // usWinDescent
02067 // HACK: All reasonable code pages are marked functional for convenience.
02068 cacheU32 (os2, 0x603f01ff); // ulCodePageRange1
02069 cacheU32 (os2, 0xffff0000); // ulCodePageRange2
02070 cacheU16 (os2, FU (8)); // sxHeight
02071 cacheU16 (os2, FU (10)); // sCapHeight
02072 cacheU16 (os2, 0); // usDefaultChar
02073 cacheU16 (os2, 0x20); // usBreakChar
02074 cacheU16 (os2, 0); // usMaxContext
02075 cacheU16 (os2, 0); // usLowerOpticalPointSize
02076 cacheU16 (os2, 0xffff); // usUpperOpticalPointSize
02077 }

```

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](#).

```

02219 {
02220     Buffer *post = newBuffer (32);
02221     addTable (font, "post", post);
02222     cacheU32 (post, 0x00030000); // version = 3.0
02223     cacheU32 (post, 0); // italicAngle
02224     cacheU16 (post, 0); // underlinePosition
02225     cacheU16 (post, 1); // underlineThickness
02226     cacheU32 (post, 1); // isFixedPitch

```



```

02227     cacheU32 (post, 0); // minMemType42
02228     cacheU32 (post, 0); // maxMemType42
02229     cacheU32 (post, 0); // minMemType1
02230     cacheU32 (post, 0); // maxMemType1
02231 }

```

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](#).

```

01599 {
01600     Buffer *glyf = newBuffer (65536);
01601     addTable (font, "glyf", glyf);
01602     Buffer *loca = newBuffer (4 * (font->glyphCount + 1));
01603     addTable (font, "loca", loca);
01604     *format = LOCA_OFFSET32;
01605     Buffer *endPoints = newBuffer (256);
01606     Buffer *flags = newBuffer (256);
01607     Buffer *xs = newBuffer (256);
01608     Buffer *ys = newBuffer (256);
01609     Buffer *outline = newBuffer (1024);
01610     Glyph *const glyphs = getBufferHead (font->glyphs);
01611     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
01612     for (Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
01613     {
01614         cacheU32 (loca, countBufferedBytes (glyf));
01615         pixels_t rx = -glyph->pos;
01616         pixels_t ry = DESCENDER;
01617         pixels_t xMin = GLYPH_MAX_WIDTH, xMax = 0;
01618         pixels_t yMin = ASCENDER, yMax = -DESCENDER;
01619         resetBuffer (endPoints);
01620         resetBuffer (flags);
01621         resetBuffer (xs);
01622         resetBuffer (ys);
01623         resetBuffer (outline);
01624         buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_RIGHT);
01625         uint_fast32_t pointCount = 0, contourCount = 0;
01626         for (const pixels_t *p = getBufferHead (outline),
01627              *const end = getBufferTail (outline); p < end;)
01628         {
01629             const enum ContourOp op = *p++;
01630             if (op == OP_CLOSE)
01631             {
01632                 contourCount++;
01633                 assert (contourCount <= U16MAX);
01634                 cacheU16 (endPoints, pointCount - 1);
01635                 continue;
01636             }
01637             assert (op == OP_POINT);
01638             pointCount++;

```

```

01639     assert (pointCount <= U16MAX);
01640     const pixels_t x = *p++, y = *p++;
01641     uint_fast8_t pointFlags =
01642         + B1 (0) // point is on curve
01643         + BX (1, x != rx) // x coordinate is 1 byte instead of 2
01644         + BX (2, y != ry) // y coordinate is 1 byte instead of 2
01645         + B0 (3) // repeat
01646         + BX (4, x >= rx) // when x is 1 byte: x is positive;
01647           // when x is 2 bytes: x unchanged and omitted
01648         + BX (5, y >= ry) // when y is 1 byte: y is positive;
01649           // when y is 2 bytes: y unchanged and omitted
01650         + B1 (6) // contours may overlap
01651         + B0 (7) // reserved
01652     ;
01653     cacheU8 (flags, pointFlags);
01654     if (x != rx)
01655         cacheU8 (xs, FU (x > rx ? x - rx : rx - x));
01656     if (y != ry)
01657         cacheU8 (ys, FU (y > ry ? y - ry : ry - y));
01658     if (x < xMin) xMin = x;
01659     if (y < yMin) yMin = y;
01660     if (x > xMax) xMax = x;
01661     if (y > yMax) yMax = y;
01662     rx = x;
01663     ry = y;
01664 }
01665 if (contourCount == 0)
01666     continue; // blank glyph is indicated by the 'loca' table
01667 glyph->lsb = glyph->pos + xMin;
01668 cacheU16 (glyf, contourCount); // numberOfContours
01669 cacheU16 (glyf, FU (glyph->pos + xMin)); // xMin
01670 cacheU16 (glyf, FU (yMin)); // yMin
01671 cacheU16 (glyf, FU (glyph->pos + xMax)); // xMax
01672 cacheU16 (glyf, FU (yMax)); // yMax
01673 cacheBuffer (glyf, endPoints); // endPtsOfContours[]
01674 cacheU16 (glyf, 0); // instructionLength
01675 cacheBuffer (glyf, flags); // flags[]
01676 cacheBuffer (glyf, xs); // xCoordinates[]
01677 cacheBuffer (glyf, ys); // yCoordinates[]
01678 if (pointCount > *maxPoints)
01679     *maxPoints = pointCount;
01680 if (contourCount > *maxContours)
01681     *maxContours = contourCount;
01682 }
01683 cacheU32 (loca, countBufferedBytes (glyf));
01684 freeBuffer (endPoints);
01685 freeBuffer (flags);
01686 freeBuffer (xs);
01687 freeBuffer (ys);
01688 freeBuffer (outline);
01689 }

```

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 <code>Buffer *</code> .
----	-----	---

Definition at line 337 of file `hex2otf.c`.

```

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

```

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](#).

```

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

```

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

5.3.5.33 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](#).

```

02604 {
02605     initBuffers (16);
02606     atexit (cleanBuffers);
02607     Options opt = parseOptions (argv);
02608     Font font;
02609     font.tables = newBuffer (sizeof (Table) * 16);
02610     font.glyphs = newBuffer (sizeof (Glyph) * MAX_GLYPHS);
02611     readGlyphs (&font, opt.hex);
02612     sortGlyphs (&font);
02613     enum LocaFormat loca = LOCA_OFFSET16;
02614     uint_fast16_t maxPoints = 0, maxContours = 0;
02615     pixels_t xMin = 0;
02616     if (opt.pos)
02617         positionGlyphs (&font, opt.pos, &xMin);
02618     if (opt.gpos)
02619         fillGposTable (&font);
02620     if (opt.gsub)
02621         fillGsubTable (&font);
02622     if (opt.cff)
02623         fillCFF (&font, opt.cff, opt.nameStrings);
02624     if (opt.truetype)
02625         fillTrueType (&font, &loca, &maxPoints, &maxContours);
02626     if (opt.blankOutline)
02627         fillBlankOutline (&font);
02628     if (opt.bitmap)
02629         fillBitmap (&font);
02630     fillHeadTable (&font, loca, xMin);
02631     fillHheaTable (&font, xMin);
02632     fillMaxpTable (&font, opt.cff, maxPoints, maxContours);
02633     fillOS2Table (&font);
02634     fillNameTable (&font, opt.nameStrings);
02635     fillHmtxTable (&font);
02636     fillCmapTable (&font);
02637     fillPostTable (&font);
02638     organizeTables (&font, opt.cff);
02639     writeFont (&font, opt.cff, opt.out);
02640     return EXIT_SUCCESS;
02641 }

```

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	delimiter	The delimiter to end searching.

Returns

Pointer to the first character of the desired option.

Definition at line 2470 of file [hex2otf.c](#).

```

02471 {
02472     while (*key)

```

```

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

```

Here is the caller graph for this function:

5.3.5.35 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](#).

```

00189 {
00190     assert (initialCapacity > 0);
00191     Buffer *buf = NULL;
00192     size_t sentinel = nextBufferIndex;
00193     do
00194     {
00195         if (nextBufferIndex == bufferCount)
00196             nextBufferIndex = 0;
00197         if (allBuffers[nextBufferIndex].capacity == 0)
00198         {
00199             buf = &allBuffers[nextBufferIndex++];
00200             break;
00201         }
00202     } while (++nextBufferIndex != sentinel);
00203     if (!buf) // no existing buffer available
00204     {
00205         size_t newSize = sizeof (Buffer) * bufferCount * 2;
00206         void *extended = realloc (allBuffers, newSize);
00207         if (!extended)
00208             fail ("Failed to create new buffers.");
00209         allBuffers = extended;
00210         memset (allBuffers + bufferCount, 0, sizeof (Buffer) * bufferCount);
00211         buf = &allBuffers[bufferCount];
00212         nextBufferIndex = bufferCount + 1;
00213         bufferCount *= 2;
00214     }
00215     buf->begin = malloc (initialCapacity);
00216     if (!buf->begin)
00217         fail ("Failed to allocate %zu bytes of memory.", initialCapacity);
00218     buf->capacity = initialCapacity;
00219     buf->next = buf->begin;
00220     buf->end = buf->begin + initialCapacity;
00221     return buf;
00222 }

```

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](#).

```

00712 {
00713     const char *const cffOrder[] = {"head","hhea","maxp","OS/2","name",
00714         "cmap","post","CFF ",NULL};
00715     const char *const truetypeOrder[] = {"head","hhea","maxp","OS/2",
00716         "hmtx","LTSH","VDMX","hdmx","cmap","fpgm","prep","cvt ","loca",
00717         "glyf","kern","name","post","gasp","PCLT","DSIG",NULL};
00718     const char *const *const order = isCFF ? cffOrder : truetypeOrder;
00719     Table *unordered = getBufferHead (font->tables);
00720     const Table *const tablesEnd = getBufferTail (font->tables);
00721     for (const char *const *p = order; *p; p++)
00722     {
00723         uint_fast32_t tag = tagAsU32 (*p);
00724         for (Table *t = unordered; t < tablesEnd; t++)
00725         {
00726             if (t->tag != tag)
00727                 continue;
00728             if (t != unordered)
00729             {
00730                 Table temp = *unordered;
00731                 *unordered = *t;
00732                 *t = temp;
00733             }
00734             unordered++;
00735             break;
00736         }
00737     }
00738 }
```

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](#).

```

02501 {
02502     Options opt = {0}; // all options default to 0, false and NULL
02503     const char *format = NULL;
02504     struct StringArg
02505     {
02506         const char *const key;
02507         const char **const value;
02508     } strArgs[] =
02509     {
02510         {"hex", &opt.hex},
02511         {"pos", &opt.pos},
02512         {"out", &opt.out},
02513         {"format", &format},
02514         {NULL, NULL} // sentinel
02515     };
02516     for (char *const *argp = argv + 1; *argp; argp++)
02517     {
02518         const char *const arg = *argp;
02519         struct StringArg *p;
02520         const char *value = NULL;
02521         if (strcmp (arg, "--help") == 0)
02522             printHelp ();
02523         if (strcmp (arg, "--version") == 0)
02524             printVersion ();
02525         for (p = strArgs; p->key; p++)
02526             if ((value = matchToken (arg, p->key, '=')))
02527                 break;
02528         if (p->key)
02529         {
02530             if (!*value)
02531                 fail ("Empty argument: '%s'", p->key);
02532             if (*p->value)
02533                 fail ("Duplicate argument: '%s'", p->key);
02534             *p->value = value;
02535         }
02536         else // shall be a name string
02537         {
02538             char *endptr;
02539             unsigned long id = strtoul (arg, &endptr, 10);
02540             if (endptr == arg || id >= MAX_NAME_IDS || *endptr != '=')
02541                 fail ("Invalid argument: '%s'", arg);
02542             endptr++; // skip '='
02543             if (opt.nameStrings[id])
02544                 fail ("Duplicate name ID: %lu.", id);
02545             opt.nameStrings[id] = endptr;
02546         }
02547     }
02548     if (!opt.hex)
02549         fail ("Hex file is not specified.");
02550     if (opt.pos && opt.pos[0] == '\0')
02551         opt.pos = NULL; // Position file is optional. Empty path means none.
02552     if (!opt.out)
02553         fail ("Output file is not specified.");
02554     if (!format)
02555         fail ("Format is not specified.");
02556     for (const NamePair *p = defaultNames; p->str; p++)
02557         if (!opt.nameStrings[p->id])
02558             opt.nameStrings[p->id] = p->str;
02559     bool cff = false, cff2 = false;
02560     struct Symbol
02561     {
02562         const char *const key;
02563         bool *const found;
02564     } symbols[] =
02565     {

```

```

02566     {"cff", &cff},
02567     {"cff2", &cff2},
02568     {"truetype", &opt.truetype},
02569     {"blank", &opt.blankOutline},
02570     {"bitmap", &opt.bitmap},
02571     {"gpos", &opt.gpos},
02572     {"gsub", &opt.gsub},
02573     {NULL, NULL} // sentinel
02574 };
02575 while (*format)
02576 {
02577     const struct Symbol *p;
02578     const char *next = NULL;
02579     for (p = symbols; p->key; p++)
02580         if ((next = matchToken (format, p->key, ',')))
02581             break;
02582     if (!p->key)
02583         fail ("Invalid format.");
02584     *p->found = true;
02585     format = next;
02586 }
02587 if (cff + cff2 + opt.truetype + opt.blankOutline > 1)
02588     fail ("At most one outline format can be accepted.");
02589 if (!(cff || cff2 || opt.truetype || opt.bitmap))
02590     fail ("Invalid format.");
02591 opt.cff = cff + cff2 * 2;
02592 return opt;
02593 }

```

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](#).

```

01062 {
01063     *xMin = 0;
01064     FILE *file = fopen (fileName, "r");
01065     if (!file)
01066         fail ("Failed to open file '%s'", fileName);
01067     Glyph *glyphs = getBufferHead (font->glyphs);
01068     const Glyph *const endGlyph = glyphs + font->glyphCount;
01069     Glyph *nextGlyph = &glyphs[1]; // predict and avoid search
01070     for (;;)
01071     {
01072         uint_fast32_t codePoint;

```



```

01073     if (readCodePoint (&codePoint, fileName, file))
01074         break;
01075     Glyph *glyph = nextGlyph;
01076     if (glyph == endGlyph || glyph->codePoint != codePoint)
01077     {
01078         // Prediction failed. Search.
01079         const Glyph key = { .codePoint = codePoint };
01080         glyph = bsearch (&key, glyphs + 1, font->glyphCount - 1,
01081             sizeof key, byCodePoint);
01082         if (!glyph)
01083             fail ("Glyph \"PRI_CP\" is positioned but not defined.",
01084                 codePoint);
01085     }
01086     nextGlyph = glyph + 1;
01087     char s[8];
01088     if (!fgets (s, sizeof s, file))
01089         fail ("%s: Read error.", fileName);
01090     char *end;
01091     const long value = strtol (s, &end, 10);
01092     if (*end != '\n' && *end != '\0')
01093         fail ("Position of glyph \"PRI_CP\" is invalid.", codePoint);
01094     // Currently no glyph is moved to the right,
01095     // so positive position is considered out of range.
01096     // If this limit is to be lifted,
01097     // 'xMax' of bounding box in 'head' table shall also be updated.
01098     if (value < -GLYPH_MAX_WIDTH || value > 0)
01099         fail ("Position of glyph \"PRI_CP\" is out of range.", codePoint);
01100     glyph->combining = true;
01101     glyph->pos = value;
01102     glyph->lsb = value; // updated during outline generation
01103     if (value < *xMin)
01104         *xMin = value;
01105 }
01106 fclose (file);
01107 }

```

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](#).

```

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

```

Here is the call graph for this function: 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 > 0xffffffff);
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 (
    void )
```

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");
02441     printf ("For more information, consult the hex2otf(1) man page.\n\n");
02442     exit (EXIT_SUCCESS);
02443 }
02444 }
```

Here is the caller graph for this function:

5.3.5.42 printVersion()

```
void printVersion (
    void )
```

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)
00927     {
00928         *codePoint = (*codePoint « 4) | nibbleValue (c);
00929         continue;
00930     }
00931     if (c == ':' && digitCount > 0)
00932         return false;
00933     if (c == EOF)
00934     {
00935         if (digitCount == 0)
00936             return true;
00937         if (feof (file))
00938             fail ("%s: Unexpected end of file.", fileName);
00939         else
00940             fail ("%s: Read error.", fileName);
00941     }
00942     fail ("%s: Unexpected character: %#.2x.", fileName, (unsigned)c);
00943 }
00944 }

```

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.

```

00967 {
00968     FILE *file = fopen (fileName, "r");
00969     if (!file)
00970         fail ("Failed to open file '%s'", fileName);
00971     uint_fast32_t glyphCount = 1; // for glyph 0
00972     uint_fast8_t maxByteCount = 0;
00973     { // Hard code the .notdef glyph.
00974         const byte bitmap[] = "\0\0\0-fZZzvv-vv-\0\0"; // same as U+FFFD
00975         const size_t byteCount = sizeof bitmap - 1;
00976         assert (byteCount <= GLYPH_MAX_BYTE_COUNT);
00977         assert (byteCount % GLYPH_HEIGHT == 0);
00978         Glyph *notdef = getBufferSlot (font->glyphs, sizeof (Glyph));
00979         memcpy (notdef->bitmap, bitmap, byteCount);

```

```

00980     notdef->byteCount = maxByteCount = byteCount;
00981     notdef->combining = false;
00982     notdef->pos = 0;
00983     notdef->lsb = 0;
00984 }
00985 for (;;)
00986 {
00987     uint_fast32_t codePoint;
00988     if (readCodePoint (&codePoint, fileName, file))
00989         break;
00990     if (++glyphCount > MAX_GLYPHS)
00991         fail ("OpenType does not support more than %lu glyphs.",
00992             MAX_GLYPHS);
00993     Glyph *glyph = getBufferSlot (font->glyphs, sizeof (Glyph));
00994     glyph->codePoint = codePoint;
00995     glyph->byteCount = 0;
00996     glyph->combining = false;
00997     glyph->pos = 0;
00998     glyph->lsb = 0;
00999     for (byte *p = glyph->bitmap; p++)
01000     {
01001         int h, l;
01002         if (isxdigit (h = getc (file)) && isxdigit (l = getc (file)))
01003         {
01004             if (++glyph->byteCount > GLYPH_MAX_BYTE_COUNT)
01005                 fail ("Hex stream of \"PRI_CP\" is too long.", codePoint);
01006             *p = nibbleValue (h) « 4 | nibbleValue (l);
01007         }
01008         else if (h == '\n' || (h == EOF && feof (file)))
01009             break;
01010         else if (ferror (file))
01011             fail ("%s: Read error.", fileName);
01012         else
01013             fail ("Hex stream of \"PRI_CP\" is invalid.", codePoint);
01014     }
01015     if (glyph->byteCount % GLYPH_HEIGHT != 0)
01016         fail ("Hex length of \"PRI_CP\" is indivisible by glyph height %d.",
01017             codePoint, GLYPH_HEIGHT);
01018     if (glyph->byteCount > maxByteCount)
01019         maxByteCount = glyph->byteCount;
01020 }
01021 if (glyphCount == 1)
01022     fail ("No glyph is specified.");
01023 font->glyphCount = glyphCount;
01024 font->maxWidth = PW (maxByteCount);
01025 fclose (file);
01026 }

```

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](#).

```

01120 {
01121     Glyph *glyphs = getBufferHead (font->glyphs);

```

```

01122     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
01123     glyphs++; // glyph 0 does not need sorting
01124     qsort (glyphs, glyphsEnd - glyphs, sizeof *glyphs, byCodePoint);
01125     for (const Glyph *glyph = glyphs; glyph < glyphsEnd - 1; glyph++)
01126     {
01127         if (glyph[0].codePoint == glyph[1].codePoint)
01128             fail ("Duplicate code point: "PRI_CP", glyph[0].codePoint);
01129         assert (glyph[0].codePoint < glyph[1].codePoint);
01130     }
01131 }

```

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](#).

```

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

```

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

5.3.5.47 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 }
00867 freeBuffer (tableRecords);
00868 for (const Table *table = tables; table < tablesEnd; table++)
00869 {
00870     if (table->tag == 0x68656164) // 'head' table
00871     {
00872         byte *begin = getBufferHead (table->content);
00873         byte *end = getBufferTail (table->content);
00874         writeBytes (begin, 8, file);
00875         writeU32 (0xb1b0afbaU - totalChecksum, file); // checksumAdjustment
00876         writeBytes (begin + 12, end - (begin + 12), file);
00877         continue;
00878     }
00879     writeBuffer (table->content, file);
00880 }
00881 fclose (file);
00882 }

```

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.](#)

```

00001 /**
00002  @file hex2otf.c
00003
00004  @brief hex2otf - Convert GNU Unifont .hex file to OpenType font
00005
00006  This program reads a Unifont .hex format file and a file containing
00007  combining mark offset information, and produces an OpenType font file.
00008
00009  @copyright Copyright © 2022 何志翔 (He Zhixiang)
00010
00011  @author 何志翔 (He Zhixiang)
00012 */
00013
00014 /*
00015  LICENSE:
00016
00017  This program is free software; you can redistribute it and/or
00018  modify it under the terms of the GNU General Public License
00019  as published by the Free Software Foundation; either version 2
00020  of the License, or (at your option) any later version.
00021
00022  This program is distributed in the hope that it will be useful,
00023  but WITHOUT ANY WARRANTY; without even the implied warranty of
00024  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00025  GNU General Public License for more details.
00026
00027  You should have received a copy of the GNU General Public License
00028  along with this program; if not, write to the Free Software
00029  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
00030  02110-1301, USA.
00031
00032  NOTE: It is a violation of the license terms of this software
00033  to delete or override license and copyright information contained
00034  in the hex2otf.h file if creating a font derived from Unifont glyphs.
00035  Fonts derived from Unifont can add names to the copyright notice
00036  for creators of new or modified glyphs.
00037 */
00038
00039 #include <assert.h>
00040 #include <ctype.h>
00041 #include <inttypes.h>
00042 #include <stdarg.h>
00043 #include <stdbool.h>
00044 #include <stddef.h>
00045 #include <stdio.h>
00046 #include <stdlib.h>
00047 #include <string.h>
00048
00049 #include "hex2otf.h"
00050
00051 #define VERSION "1.0.1" ///< Program version, for "--version" option.
00052
00053 // This program assumes the execution character set is compatible with ASCII.
00054
00055 #define U16MAX 0xffff ///< Maximum UTF-16 code point value.
00056 #define U32MAX 0xffffffff ///< Maximum UTF-32 code point value.
00057
00058 #define PRI_CP "U+%.*"PRIXFAST32 ///< Format string to print Unicode code point.
00059
00060 #ifndef static_assert
00061 #define static_assert(a, b) (assert(a)) ///< If "a" is true, return string "b".
00062 #endif
00063
00064 // Set or clear a particular bit.
00065 #define BX(shift, x) ((uintmax_t)(!!(x)) << (shift)) ///< Truncate & shift word.
00066 #define B0(shift) BX((shift), 0) ///< Clear a given bit in a word.
00067 #define B1(shift) BX((shift), 1) ///< Set a given bit in a word.
00068
00069 #define GLYPH_MAX_WIDTH 16 ///< Maximum glyph width, in pixels.
00070 #define GLYPH_HEIGHT 16 ///< Maximum glyph height, in pixels.
00071
00072 /// Number of bytes to represent one bitmap glyph as a binary array.
00073 #define GLYPH_MAX_BYTE_COUNT (GLYPH_HEIGHT * GLYPH_MAX_WIDTH / 8)
00074
00075 /// Count of pixels below baseline.
00076 #define DESCENDER 2

```

```

00077
00078 /// Count of pixels above baseline.
00079 #define ASCENDER (GLYPH_HEIGHT - DESCENDER)
00080
00081 /// Font units per em.
00082 #define FUPEM 64
00083
00084 /// An OpenType font has at most 65536 glyphs.
00085 #define MAX_GLYPHS 65536
00086
00087 /// Name IDs 0-255 are used for standard names.
00088 #define MAX_NAME_IDS 256
00089
00090 /// Convert pixels to font units.
00091 #define FU(x) ((x) * FUPEM / GLYPH_HEIGHT)
00092
00093 /// Convert glyph byte count to pixel width.
00094 #define PW(x) ((x) / (GLYPH_HEIGHT / 8))
00095
00096 /// Definition of "byte" type as an unsigned char.
00097 typedef unsigned char byte;
00098
00099 /// This type must be able to represent max(GLYPH_MAX_WIDTH, GLYPH_HEIGHT).
00100 typedef int_least8_t pixels_t;
00101
00102 /**
00103  * @brief Print an error message on stderr, then exit.
00104  *
00105  * This function prints the provided error string and optional
00106  * following arguments to stderr, and then exits with a status
00107  * of EXIT_FAILURE.
00108  *
00109  * @param[in] reason The output string to describe the error.
00110  * @param[in] ... Optional following arguments to output.
00111  */
00112 void
00113 fail (const char *reason, ...)
00114 {
00115     fputs ("ERROR: ", stderr);
00116     va_list args;
00117     va_start (args, reason);
00118     vfprintf (stderr, reason, args);
00119     va_end (args);
00120     putc ('\n', stderr);
00121     exit (EXIT_FAILURE);
00122 }
00123
00124 /**
00125  * @brief Generic data structure for a linked list of buffer elements.
00126  *
00127  * A buffer can act as a vector (when filled with 'store*' functions),
00128  * or a temporary output area (when filled with 'cache*' functions).
00129  * The 'store*' functions use native endian.
00130  * The 'cache*' functions use big endian or other formats in OpenType.
00131  * Beware of memory alignment.
00132  */
00133 typedef struct Buffer
00134 {
00135     size_t capacity; // = 0 iff this buffer is free
00136     byte *begin, *next, *end;
00137 } Buffer;
00138
00139 Buffer *allBuffers; ///< Initial allocation of empty array of buffer pointers.
00140 size_t bufferCount; ///< Number of buffers in a Buffer * array.
00141 size_t nextBufferIndex; ///< Index number to tail element of Buffer * array.
00142
00143 /**
00144  * @brief Initialize an array of buffer pointers to all zeroes.
00145  *
00146  * This function initializes the "allBuffers" array of buffer
00147  * pointers to all zeroes.
00148  *
00149  * @param[in] count The number of buffer array pointers to allocate.
00150  */
00151 void
00152 initBuffers (size_t count)
00153 {
00154     assert (count > 0);
00155     assert (bufferCount == 0); // uninitialized
00156     allBuffers = calloc (count, sizeof *allBuffers);
00157     if (!allBuffers)

```

```

00158     fail ("Failed to initialize buffers.");
00159     bufferCount = count;
00160     nextBufferIndex = 0;
00161 }
00162
00163 /**
00164     @brief Free all allocated buffer pointers.
00165
00166     This function frees all buffer pointers previously allocated
00167     in the initBuffers function.
00168 */
00169 void
00170 cleanBuffers (void)
00171 {
00172     for (size_t i = 0; i < bufferCount; i++)
00173         if (allBuffers[i].capacity)
00174             free (allBuffers[i].begin);
00175     free (allBuffers);
00176     bufferCount = 0;
00177 }
00178
00179 /**
00180     @brief Create a new buffer.
00181
00182     This function creates a new buffer array of type Buffer,
00183     with an initial size of initialCapacity elements.
00184
00185     @param[in] initialCapacity The initial number of elements in the buffer.
00186 */
00187 Buffer *
00188 newBuffer (size_t initialCapacity)
00189 {
00190     assert (initialCapacity > 0);
00191     Buffer *buf = NULL;
00192     size_t sentinel = nextBufferIndex;
00193     do
00194     {
00195         if (nextBufferIndex == bufferCount)
00196             nextBufferIndex = 0;
00197         if (allBuffers[nextBufferIndex].capacity == 0)
00198         {
00199             buf = &allBuffers[nextBufferIndex++];
00200             break;
00201         }
00202     } while (++nextBufferIndex != sentinel);
00203     if (!buf) // no existing buffer available
00204     {
00205         size_t newSize = sizeof (Buffer) * bufferCount * 2;
00206         void *extended = realloc (allBuffers, newSize);
00207         if (!extended)
00208             fail ("Failed to create new buffers.");
00209         allBuffers = extended;
00210         memset (allBuffers + bufferCount, 0, sizeof (Buffer) * bufferCount);
00211         buf = &allBuffers[bufferCount];
00212         nextBufferIndex = bufferCount + 1;
00213         bufferCount *= 2;
00214     }
00215     buf->begin = malloc (initialCapacity);
00216     if (!buf->begin)
00217         fail ("Failed to allocate %zu bytes of memory.", initialCapacity);
00218     buf->capacity = initialCapacity;
00219     buf->next = buf->begin;
00220     buf->end = buf->begin + initialCapacity;
00221     return buf;
00222 }
00223
00224 /**
00225     @brief Ensure that the buffer has at least the specified minimum size.
00226
00227     This function takes a buffer array of type Buffer and the
00228     necessary minimum number of elements as inputs, and attempts
00229     to increase the size of the buffer if it must be larger.
00230
00231     If the buffer is too small and cannot be resized, the program
00232     will terminate with an error message and an exit status of
00233     EXIT_FAILURE.
00234
00235     @param[in,out] buf The buffer to check.
00236     @param[in] needed The required minimum number of elements in the buffer.
00237 */
00238 void

```

```

00239 ensureBuffer (Buffer *buf, size_t needed)
00240 {
00241     if (buf->end - buf->next >= needed)
00242         return;
00243     ptrdiff_t occupied = buf->next - buf->begin;
00244     size_t required = occupied + needed;
00245     if (required < needed) // overflow
00246         fail ("Cannot allocate %zu + %zu bytes of memory.", occupied, needed);
00247     if (required > SIZE_MAX / 2)
00248         buf->capacity = required;
00249     else while (buf->capacity < required)
00250         buf->capacity *= 2;
00251     void *extended = realloc (buf->begin, buf->capacity);
00252     if (!extended)
00253         fail ("Failed to allocate %zu bytes of memory.", buf->capacity);
00254     buf->begin = extended;
00255     buf->next = buf->begin + occupied;
00256     buf->end = buf->begin + buf->capacity;
00257 }
00258
00259 /**
00260     @brief Count the number of elements in a buffer.
00261
00262     @param[in] buf The buffer to be examined.
00263     @return The number of elements in the buffer.
00264 */
00265 static inline size_t
00266 countBufferedBytes (const Buffer *buf)
00267 {
00268     return buf->next - buf->begin;
00269 }
00270
00271 /**
00272     @brief Get the start of the buffer array.
00273
00274     @param[in] buf The buffer to be examined.
00275     @return A pointer of type Buffer * to the start of the buffer.
00276 */
00277 static inline void *
00278 getBufferHead (const Buffer *buf)
00279 {
00280     return buf->begin;
00281 }
00282
00283 /**
00284     @brief Get the end of the buffer array.
00285
00286     @param[in] buf The buffer to be examined.
00287     @return A pointer of type Buffer * to the end of the buffer.
00288 */
00289 static inline void *
00290 getBufferTail (const Buffer *buf)
00291 {
00292     return buf->next;
00293 }
00294
00295 /**
00296     @brief Add a slot to the end of a buffer.
00297
00298     This function ensures that the buffer can grow by one slot,
00299     and then returns a pointer to the new slot within the buffer.
00300
00301     @param[in] buf The pointer to an array of type Buffer *.
00302     @param[in] slotSize The new slot number.
00303     @return A pointer to the new slot within the buffer.
00304 */
00305 static inline void *
00306 getBufferSlot (Buffer *buf, size_t slotSize)
00307 {
00308     ensureBuffer (buf, slotSize);
00309     void *slot = buf->next;
00310     buf->next += slotSize;
00311     return slot;
00312 }
00313
00314 /**
00315     @brief Reset a buffer pointer to the buffer's beginning.
00316
00317     This function resets an array of type Buffer * to point
00318     its tail to the start of the array.
00319

```

```

00320  @param[in] buf The pointer to an array of type Buffer *.
00321  */
00322  static inline void
00323  resetBuffer (Buffer *buf)
00324  {
00325      buf->next = buf->begin;
00326  }
00327
00328  /**
00329   @brief Free the memory previously allocated for a buffer.
00330
00331   This function frees the memory allocated to an array
00332   of type Buffer *.
00333
00334   @param[in] buf The pointer to an array of type Buffer *.
00335  */
00336  void
00337  freeBuffer (Buffer *buf)
00338  {
00339      free (buf->begin);
00340      buf->capacity = 0;
00341  }
00342
00343  /**
00344   @brief Temporary define to look up an element in an array of given type.
00345
00346   This definition is used to create lookup functions to return
00347   a given element in unsigned arrays of size 8, 16, and 32 bytes,
00348   and in an array of pixels.
00349  */
00350  #define defineStore(name, type) \
00351  void name (Buffer *buf, type value) \
00352  { \
00353      type *slot = getBufferSlot (buf, sizeof value); \
00354      *slot = value; \
00355  }
00356  defineStore (storeU8, uint_least8_t)
00357  defineStore (storeU16, uint_least16_t)
00358  defineStore (storeU32, uint_least32_t)
00359  defineStore (storePixels, pixels_t)
00360  #undef defineStore
00361
00362  /**
00363   @brief Cache bytes in a big-endian format.
00364
00365   This function adds from 1, 2, 3, or 4 bytes to the end of
00366   a byte array in big-endian order. The buffer is updated
00367   to account for the newly-added bytes.
00368
00369   @param[in,out] buf The array of bytes to which to append new bytes.
00370   @param[in] value The bytes to add, passed as a 32-bit unsigned word.
00371   @param[in] bytes The number of bytes to append to the buffer.
00372  */
00373  void
00374  cacheU (Buffer *buf, uint_fast32_t value, int bytes)
00375  {
00376      assert (1 <= bytes && bytes <= 4);
00377      ensureBuffer (buf, bytes);
00378      switch (bytes)
00379      {
00380          case 4: *buf->next++ = value » 24 & 0xff; // fall through
00381          case 3: *buf->next++ = value » 16 & 0xff; // fall through
00382          case 2: *buf->next++ = value » 8 & 0xff; // fall through
00383          case 1: *buf->next++ = value & 0xff;
00384      }
00385  }
00386
00387  /**
00388   @brief Append one unsigned byte to the end of a byte array.
00389
00390   This function adds one byte to the end of a byte array.
00391   The buffer is updated to account for the newly-added byte.
00392
00393   @param[in,out] buf The array of bytes to which to append a new byte.
00394   @param[in] value The 8-bit unsigned value to append to the buf array.
00395  */
00396  void
00397  cacheU8 (Buffer *buf, uint_fast8_t value)
00398  {
00399      storeU8 (buf, value & 0xff);
00400  }

```

```

00401
00402 /**
00403  @brief Append two unsigned bytes to the end of a byte array.
00404
00405  This function adds two bytes to the end of a byte array.
00406  The buffer is updated to account for the newly-added bytes.
00407
00408  @param[in,out] buf The array of bytes to which to append two new bytes.
00409  @param[in] value The 16-bit unsigned value to append to the buf array.
00410 */
00411 void
00412 cacheU16 (Buffer *buf, uint_fast16_t value)
00413 {
00414     cacheU (buf, value, 2);
00415 }
00416
00417 /**
00418  @brief Append four unsigned bytes to the end of a byte array.
00419
00420  This function adds four bytes to the end of a byte array.
00421  The buffer is updated to account for the newly-added bytes.
00422
00423  @param[in,out] buf The array of bytes to which to append four new bytes.
00424  @param[in] value The 32-bit unsigned value to append to the buf array.
00425 */
00426 void
00427 cacheU32 (Buffer *buf, uint_fast32_t value)
00428 {
00429     cacheU (buf, value, 4);
00430 }
00431
00432 /**
00433  @brief Cache charstring number encoding in a CFF buffer.
00434
00435  This function caches two's complement 8-, 16-, and 32-bit
00436  words as per Adobe's Type 2 Charstring encoding for operands.
00437  These operands are used in Compact Font Format data structures.
00438
00439  Byte values can have offsets, for which this function
00440  compensates, optionally followed by additional bytes:
00441
00442      Byte Range  Offset  Bytes  Adjusted Range
00443      -----
00444      0 to 11      0      1      0 to 11 (operators)
00445      12           0      2      Next byte is 8-bit op code
00446      13 to 18     0      1      13 to 18 (operators)
00447      19 to 20     0      2+     hintmask and cntrmask operators
00448      21 to 27     0      1      21 to 27 (operators)
00449      28           0      3      16-bit 2's complement number
00450      29 to 31     0      1      29 to 31 (operators)
00451      32 to 246   -139     1      -107 to +107
00452      247 to 250  +108     2      +108 to +1131
00453      251 to 254  -108     2      -108 to -1131
00454      255         0      5      16-bit integer and 16-bit fraction
00455
00456  @param[in,out] buf The buffer to which the operand value is appended.
00457  @param[in] value The operand value.
00458 */
00459 void
00460 cacheCFFOperand (Buffer *buf, int_fast32_t value)
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 }
00483
00484 /**
00485     @brief Append 1 to 4 bytes of zeroes to a buffer, for padding.
00486
00487     @param[in,out] buf The buffer to which the operand value is appended.
00488     @param[in] count The number of bytes containing zeroes to append.
00489 */
00490 void
00491 cacheZeros (Buffer *buf, size_t count)
00492 {
00493     ensureBuffer (buf, count);
00494     memset (buf->next, 0, count);
00495     buf->next += count;
00496 }
00497
00498 /**
00499     @brief Append a string of bytes to a buffer.
00500
00501     This function appends an array of 1 to 4 bytes to the end of
00502     a buffer.
00503
00504     @param[in,out] buf The buffer to which the bytes are appended.
00505     @param[in] src The array of bytes to append to the buffer.
00506     @param[in] count The number of bytes containing zeroes to append.
00507 */
00508 void
00509 cacheBytes (Buffer *restrict buf, const void *restrict src, size_t count)
00510 {
00511     ensureBuffer (buf, count);
00512     memcpy (buf->next, src, count);
00513     buf->next += count;
00514 }
00515
00516 /**
00517     @brief Append bytes of a table to a byte buffer.
00518
00519     @param[in,out] bufDest The buffer to which the new bytes are appended.
00520     @param[in] bufSrc The bytes to append to the buffer array.
00521 */
00522 void
00523 cacheBuffer (Buffer *restrict bufDest, const Buffer *restrict bufSrc)
00524 {
00525     size_t length = countBufferedBytes (bufSrc);
00526     ensureBuffer (bufDest, length);
00527     memcpy (bufDest->next, bufSrc->begin, length);
00528     bufDest->next += length;
00529 }
00530
00531 /**
00532     @brief Write an array of bytes to an output file.
00533
00534     @param[in] bytes An array of unsigned bytes to write.
00535     @param[in] file The file pointer for writing, of type FILE *.
00536 */
00537 void
00538 writeBytes (const byte bytes[], size_t count, FILE *file)
00539 {
00540     if (fwrite (bytes, count, 1, file) != 1 && count != 0)
00541         fail ("Failed to write %zu bytes to output file.", count);
00542 }
00543
00544 /**
00545     @brief Write an unsigned 16-bit value to an output file.
00546
00547     This function writes a 16-bit unsigned value in big-endian order
00548     to an output file specified with a file pointer.
00549
00550     @param[in] value The 16-bit value to write.
00551     @param[in] file The file pointer for writing, of type FILE *.
00552 */
00553 void
00554 writeU16 (uint_fast16_t value, FILE *file)
00555 {
00556     byte bytes[] =
00557     {
00558         (value » 8) & 0xff,
00559         (value ) & 0xff,
00560     };
00561     writeBytes (bytes, sizeof bytes, file);
00562 }

```

```

00563
00564 /**
00565  @brief Write an unsigned 32-bit value to an output file.
00566
00567  This function writes a 32-bit unsigned value in big-endian order
00568  to an output file specified with a file pointer.
00569
00570  @param[in] value The 32-bit value to write.
00571  @param[in] file The file pointer for writing, of type FILE *.
00572 */
00573 void
00574 writeU32 (uint_fast32_t value, FILE *file)
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 }
00585
00586 /**
00587  @brief Write an entire buffer array of bytes to an output file.
00588
00589  This function determines the size of a buffer of bytes and
00590  writes that number of bytes to an output file specified with
00591  a file pointer. The number of bytes is determined from the
00592  length information stored as part of the Buffer * data structure.
00593
00594  @param[in] buf An array containing unsigned bytes to write.
00595  @param[in] file The file pointer for writing, of type FILE *.
00596 */
00597 static inline void
00598 writeBuffer (const Buffer *buf, FILE *file)
00599 {
00600     writeBytes (getBufferHead (buf), countBufferedBytes (buf), file);
00601 }
00602
00603 /// Array of OpenType names indexed directly by Name IDs.
00604 typedef const char *NameStrings[MAX_NAME_IDS];
00605
00606 /**
00607  @brief Data structure to hold data for one bitmap glyph.
00608
00609  This data structure holds data to represent one Unifont bitmap
00610  glyph: Unicode code point, number of bytes in its bitmap array,
00611  whether or not it is a combining character, and an offset from
00612  the glyph origin to the start of the bitmap.
00613 */
00614 typedef struct Glyph
00615 {
00616     uint_least32_t codePoint; ///< undefined for glyph 0
00617     byte bitmap[GLYPH_MAX_BYTE_COUNT]; ///< hexadecimal bitmap character array
00618     uint_least8_t byteCount; ///< length of bitmap data
00619     bool combining; ///< whether this is a combining glyph
00620     pixels_t pos; ///< number of pixels the glyph should be moved to the right
00621     ///< (negative number means moving to the left)
00622     pixels_t lsb; ///< left side bearing (x position of leftmost contour point)
00623 } Glyph;
00624
00625 /**
00626  @brief Data structure to hold information for one font.
00627 */
00628 typedef struct Font
00629 {
00630     Buffer *tables;
00631     Buffer *glyphs;
00632     uint_fast32_t glyphCount;
00633     pixels_t maxWidth;
00634 } Font;
00635
00636 /**
00637  @brief Data structure for an OpenType table.
00638
00639  This data structure contains a table tag and a pointer to the
00640  start of the buffer that holds data for this OpenType table.
00641
00642  For information on the OpenType tables and their structure, see
00643  https://docs.microsoft.com/en-us/typography/opentype/spec/otff#font-tables.

```

```

00644 */
00645 typedef struct Table
00646 {
00647     uint_fast32_t tag;
00648     Buffer *content;
00649 } Table;
00650
00651 /**
00652  @brief Index to Location ("loca") offset information.
00653
00654  This enumerated type encodes the type of offset to locations
00655  in a table. It denotes Offset16 (16-bit) and Offset32 (32-bit)
00656  offset types.
00657 */
00658 enum LocaFormat {
00659     LOCA_OFFSET16 = 0,    ///< Offset to location is a 16-bit Offset16 value
00660     LOCA_OFFSET32 = 1    ///< Offset to location is a 32-bit Offset32 value
00661 };
00662
00663 /**
00664  @brief Convert a 4-byte array to the machine's native 32-bit endian order.
00665
00666  This function takes an array of 4 bytes in big-endian order and
00667  converts it to a 32-bit word in the endian order of the native machine.
00668
00669  @param[in] tag The array of 4 bytes in big-endian order.
00670  @return The 32-bit unsigned word in a machine's native endian order.
00671 */
00672 static inline uint_fast32_t tagAsU32 (const char tag[static 4])
00673 {
00674     uint_fast32_t r = 0;
00675     r |= (tag[0] & 0xff) << 24;
00676     r |= (tag[1] & 0xff) << 16;
00677     r |= (tag[2] & 0xff) << 8;
00678     r |= (tag[3] & 0xff);
00679     return r;
00680 }
00681
00682 /**
00683  @brief Add a TrueType or OpenType table to the font.
00684
00685  This function adds a TrueType or OpenType table to a font.
00686  The 4-byte table tag is passed as an unsigned 32-bit integer
00687  in big-endian format.
00688
00689  @param[in,out] font The font to which a font table will be added.
00690  @param[in] tag The 4-byte table name.
00691  @param[in] content The table bytes to add, of type Buffer *.
00692 */
00693 void
00694 addTable (Font *font, const char tag[static 4], Buffer *content)
00695 {
00696     Table *table = getBufferSlot (font->tables, sizeof (Table));
00697     table->tag = tagAsU32 (tag);
00698     table->content = content;
00699 }
00700
00701 /**
00702  @brief Sort tables according to OpenType recommendations.
00703
00704  The various tables in a font are sorted in an order recommended
00705  for TrueType font files.
00706
00707  @param[in,out] font The font in which to sort tables.
00708  @param[in] isCFF True iff Compact Font Format (CFF) is being used.
00709 */
00710 void
00711 organizeTables (Font *font, bool isCFF)
00712 {
00713     const char *const cffOrder[] = {"head", "hhea", "maxp", "OS/2", "name",
00714                                     "cmap", "post", "CFF ", NULL};
00715     const char *const truetypeOrder[] = {"head", "hhea", "maxp", "OS/2",
00716                                           "hmtx", "LTSH", "VDMX", "hdmx", "cmap", "fpgm", "prep", "cvt ", "loca",
00717                                           "glyf", "kern", "name", "post", "gasp", "PCLT", "DSIG", NULL};
00718     const char *const order = isCFF ? cffOrder : truetypeOrder;
00719     Table *unordered = getBufferHead (font->tables);
00720     const Table *const tablesEnd = getBufferTail (font->tables);
00721     for (const char *const *p = order; *p; p++)
00722     {
00723         uint_fast32_t tag = tagAsU32 (*p);
00724         for (Table *t = unordered; t < tablesEnd; t++)

```

```

00725     {
00726         if (t->tag != tag)
00727             continue;
00728         if (t != unordered)
00729         {
00730             Table temp = *unordered;
00731             *unordered = *t;
00732             *t = temp;
00733         }
00734         unordered++;
00735         break;
00736     }
00737 }
00738 }
00739
00740 /**
00741  @brief Data structure for data associated with one OpenType table.
00742
00743  This data structure contains an OpenType table's tag, start within
00744  an OpenType font file, length in bytes, and checksum at the end of
00745  the table.
00746 */
00747 struct TableRecord
00748 {
00749     uint_least32_t tag, offset, length, checksum;
00750 };
00751
00752 /**
00753  @brief Compare tables by 4-byte unsigned table tag value.
00754
00755  This function takes two pointers to a TableRecord data structure
00756  and extracts the four-byte tag structure element for each. The
00757  two 32-bit numbers are then compared. If the first tag is greater
00758  than the first, then gt = 1 and lt = 0, and so 1 - 0 = 1 is
00759  returned. If the first is less than the second, then gt = 0 and
00760  lt = 1, and so 0 - 1 = -1 is returned.
00761
00762  @param[in] a Pointer to the first TableRecord structure.
00763  @param[in] b Pointer to the second TableRecord structure.
00764  @return 1 if the tag in "a" is greater, -1 if less, 0 if equal.
00765 */
00766 int
00767 byTableTag (const void *a, const void *b)
00768 {
00769     const struct TableRecord *const ra = a, *const rb = b;
00770     int gt = ra->tag > rb->tag;
00771     int lt = ra->tag < rb->tag;
00772     return gt - lt;
00773 }
00774
00775 /**
00776  @brief Write OpenType font to output file.
00777
00778  This function writes the constructed OpenType font to the
00779  output file named "filename".
00780
00781  @param[in] font Pointer to the font, of type Font *.
00782  @param[in] isCFF Boolean indicating whether the font has CFF data.
00783  @param[in] filename The name of the font file to create.
00784 */
00785 void
00786 writeFont (Font *font, bool isCFF, const char *fileName)
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 }
00867 freeBuffer (tableRecords);
00868 for (const Table *table = tables; table < tablesEnd; table++)
00869 {
00870     if (table->tag == 0x68656164) // 'head' table
00871     {
00872         byte *begin = getBufferHead (table->content);
00873         byte *end = getBufferTail (table->content);
00874         writeBytes (begin, 8, file);
00875         writeU32 (0xb1b0afbU - totalChecksum, file); // checksumAdjustment
00876         writeBytes (begin + 12, end - (begin + 12), file);
00877         continue;
00878     }
00879     writeBuffer (table->content, file);
00880 }
00881 fclose (file);
00882 }
00883
00884 /**
00885     @brief Convert a hexadecimal digit character to a 4-bit number.
00886

```

```

00887 This function takes a character that contains one hexadecimal digit
00888 and returns the 4-bit value (as an unsigned 8-bit value) corresponding
00889 to the hexadecimal digit.
00890
00891 @param[in] nibble The character containing one hexadecimal digit.
00892 @return The hexadecimal digit value, 0 through 15, inclusive.
00893 */
00894 static inline byte
00895 nibbleValue (char nibble)
00896 {
00897     if (isdigit (nibble))
00898         return nibble - '0';
00899     nibble = toupper (nibble);
00900     return nibble - 'A' + 10;
00901 }
00902
00903 /**
00904  @brief Read up to 6 hexadecimal digits and a colon from file.
00905
00906  This function reads up to 6 hexadecimal digits followed by
00907  a colon from a file.
00908
00909  If the end of the file is reached, the function returns true.
00910  The file name is provided to include in an error message if
00911  the end of file was reached unexpectedly.
00912
00913  @param[out] codePoint The Unicode code point.
00914  @param[in] fileName The name of the input file.
00915  @param[in] file Pointer to the input file stream.
00916  @return true if at end of file, false otherwise.
00917 */
00918 bool
00919 readCodePoint (uint_fast32_t *codePoint, const char *fileName, FILE *file)
00920 {
00921     *codePoint = 0;
00922     uint_fast8_t digitCount = 0;
00923     for (;;)
00924     {
00925         int c = getc (file);
00926         if (isxdigit (c) && ++digitCount <= 6)
00927         {
00928             *codePoint = (*codePoint « 4) | nibbleValue (c);
00929             continue;
00930         }
00931         if (c == ':' && digitCount > 0)
00932             return false;
00933         if (c == EOF)
00934         {
00935             if (digitCount == 0)
00936                 return true;
00937             if (feof (file))
00938                 fail ("%s: Unexpected end of file.", fileName);
00939             else
00940                 fail ("%s: Read error.", fileName);
00941         }
00942         fail ("%s: Unexpected character: %#.2x.", fileName, (unsigned)c);
00943     }
00944 }
00945
00946 /**
00947  @brief Read glyph definitions from a Unifont .hex format file.
00948
00949  This function reads in the glyph bitmaps contained in a Unifont
00950  .hex format file. These input files contain one glyph bitmap
00951  per line. Each line is of the form
00952
00953      <hexadecimal code point> ':' <hexadecimal bitmap sequence>
00954
00955  The code point field typically consists of 4 hexadecimal digits
00956  for a code point in Unicode Plane 0, and 6 hexadecimal digits for
00957  code points above Plane 0. The hexadecimal bitmap sequence is
00958  32 hexadecimal digits long for a glyph that is 8 pixels wide by
00959  16 pixels high, and 64 hexadecimal digits long for a glyph that
00960  is 16 pixels wide by 16 pixels high.
00961
00962  @param[in,out] font The font data structure to update with new glyphs.
00963  @param[in] fileName The name of the Unifont .hex format input file.
00964 */
00965 void
00966 readGlyphs (Font *font, const char *fileName)
00967 {

```

```

00968 FILE *file = fopen (fileName, "r");
00969 if (!file)
00970     fail ("Failed to open file '%s'", fileName);
00971 uint_fast32_t glyphCount = 1; // for glyph 0
00972 uint_fast8_t maxByteCount = 0;
00973 { // Hard code the .notdef glyph.
00974     const byte bitmap[] = "\0\0\0~fZZzvv~vv~\0\0"; // same as U+FFFD
00975     const size_t byteCount = sizeof bitmap - 1;
00976     assert (byteCount <= GLYPH_MAX_BYTE_COUNT);
00977     assert (byteCount % GLYPH_HEIGHT == 0);
00978     Glyph *notdef = getBufferSlot (font->glyphs, sizeof (Glyph));
00979     memcpy (notdef->bitmap, bitmap, byteCount);
00980     notdef->byteCount = maxByteCount = byteCount;
00981     notdef->combining = false;
00982     notdef->pos = 0;
00983     notdef->lsb = 0;
00984 }
00985 for (;;)
00986 {
00987     uint_fast32_t codePoint;
00988     if (readCodePoint (&codePoint, fileName, file))
00989         break;
00990     if (++glyphCount > MAX_GLYPHS)
00991         fail ("OpenType does not support more than %lu glyphs",
00992             MAX_GLYPHS);
00993     Glyph *glyph = getBufferSlot (font->glyphs, sizeof (Glyph));
00994     glyph->codePoint = codePoint;
00995     glyph->byteCount = 0;
00996     glyph->combining = false;
00997     glyph->pos = 0;
00998     glyph->lsb = 0;
00999     for (byte *p = glyph->bitmap; p++)
01000     {
01001         int h, l;
01002         if (isxdigit (h = getc (file)) && isxdigit (l = getc (file)))
01003         {
01004             if (++glyph->byteCount > GLYPH_MAX_BYTE_COUNT)
01005                 fail ("Hex stream of \"PRI_CP\" is too long.", codePoint);
01006             *p = nibbleValue (h) « 4 | nibbleValue (l);
01007         }
01008         else if (h == '\n' || (h == EOF && feof (file)))
01009             break;
01010         else if (ferror (file))
01011             fail ("%s: Read error.", fileName);
01012         else
01013             fail ("Hex stream of \"PRI_CP\" is invalid.", codePoint);
01014     }
01015     if (glyph->byteCount % GLYPH_HEIGHT != 0)
01016         fail ("Hex length of \"PRI_CP\" is indivisible by glyph height %d.",
01017             codePoint, GLYPH_HEIGHT);
01018     if (glyph->byteCount > maxByteCount)
01019         maxByteCount = glyph->byteCount;
01020 }
01021 if (glyphCount == 1)
01022     fail ("No glyph is specified.");
01023 font->glyphCount = glyphCount;
01024 font->maxWidth = PW (maxByteCount);
01025 fclose (file);
01026 }
01027
01028 /**
01029  @brief Compare two Unicode code points to determine which is greater.
01030
01031  This function compares the Unicode code points contained within
01032  two Glyph data structures. The function returns 1 if the first
01033  code point is greater, and -1 if the second is greater.
01034
01035  @param[in] a A Glyph data structure containing the first code point.
01036  @param[in] b A Glyph data structure containing the second code point.
01037  @return 1 if the code point a is greater, -1 if less, 0 if equal.
01038  */
01039 int
01040 byCodePoint (const void *a, const void *b)
01041 {
01042     const Glyph *const ga = a, *const gb = b;
01043     int gt = ga->codePoint > gb->codePoint;
01044     int lt = ga->codePoint < gb->codePoint;
01045     return gt - lt;
01046 }
01047
01048 /**

```

```

01049  @brief Position a glyph within a 16-by-16 pixel bounding box.
01050
01051  Position a glyph within the 16-by-16 pixel drawing area and
01052  note whether or not the glyph is a combining character.
01053
01054  N.B.: Glyphs must be sorted by code point before calling this function.
01055
01056  @param[in,out] font Font data structure pointer to store glyphs.
01057  @param[in] fileName Name of glyph file to read.
01058  @param[in] xMin Minimum x-axis value (for left side bearing).
01059 */
01060 void
01061 positionGlyphs (Font *font, const char *fileName, pixels_t *xMin)
01062 {
01063     *xMin = 0;
01064     FILE *file = fopen (fileName, "r");
01065     if (!file)
01066         fail ("Failed to open file '%s'", fileName);
01067     Glyph *glyphs = getBufferHead (font->glyphs);
01068     const Glyph *const endGlyph = glyphs + font->glyphCount;
01069     Glyph *nextGlyph = &glyphs[1]; // predict and avoid search
01070     for (;;)
01071     {
01072         uint_fast32_t codePoint;
01073         if (readCodePoint (&codePoint, fileName, file))
01074             break;
01075         Glyph *glyph = nextGlyph;
01076         if (glyph == endGlyph || glyph->codePoint != codePoint)
01077         {
01078             // Prediction failed. Search.
01079             const Glyph key = { .codePoint = codePoint };
01080             glyph = bsearch (&key, glyphs + 1, font->glyphCount - 1,
01081                             sizeof key, byCodePoint);
01082             if (!glyph)
01083                 fail ("Glyph \"PRI_CP\" is positioned but not defined.",
01084                     codePoint);
01085         }
01086         nextGlyph = glyph + 1;
01087         char s[8];
01088         if (!fgets (s, sizeof s, file))
01089             fail ("%s: Read error.", fileName);
01090         char *end;
01091         const long value = strtol (s, &end, 10);
01092         if (*end != '\n' && *end != '\0')
01093             fail ("Position of glyph \"PRI_CP\" is invalid.", codePoint);
01094         // Currently no glyph is moved to the right,
01095         // so positive position is considered out of range.
01096         // If this limit is to be lifted,
01097         // 'xMax' of bounding box in 'head' table shall also be updated.
01098         if (value < -GLYPH_MAX_WIDTH || value > 0)
01099             fail ("Position of glyph \"PRI_CP\" is out of range.", codePoint);
01100         glyph->combining = true;
01101         glyph->pos = value;
01102         glyph->lsb = value; // updated during outline generation
01103         if (value < *xMin)
01104             *xMin = value;
01105     }
01106     fclose (file);
01107 }
01108
01109 /**
01110  @brief Sort the glyphs in a font by Unicode code point.
01111
01112  This function reads in an array of glyphs and sorts them
01113  by Unicode code point. If a duplicate code point is encountered,
01114  that will result in a fatal error with an error message to stderr.
01115
01116  @param[in,out] font Pointer to a Font structure with glyphs to sort.
01117 */
01118 void
01119 sortGlyphs (Font *font)
01120 {
01121     Glyph *glyphs = getBufferHead (font->glyphs);
01122     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
01123     glyphs++; // glyph 0 does not need sorting
01124     qsort (glyphs, glyphsEnd - glyphs, sizeof *glyphs, byCodePoint);
01125     for (const Glyph *glyph = glyphs; glyph < glyphsEnd - 1; glyph++)
01126     {
01127         if (glyph[0].codePoint == glyph[1].codePoint)
01128             fail ("Duplicate code point: \"PRI_CP\".", glyph[0].codePoint);
01129         assert (glyph[0].codePoint < glyph[1].codePoint);
01130     }
01131 }

```



```

01130     }
01131 }
01132
01133 /**
01134  @brief Specify the current contour drawing operation.
01135 */
01136 enum ContourOp {
01137     OP_CLOSE,    ///< Close the current contour path that was being drawn.
01138     OP_POINT     ///< Add one more (x,y) point to the contour being drawn.
01139 };
01140
01141 /**
01142  @brief Fill to the left side (CFF) or right side (TrueType) of a contour.
01143 */
01144 enum FillSide {
01145     FILL_LEFT,   ///< Draw outline counter-clockwise (CFF, PostScript).
01146     FILL_RIGHT   ///< Draw outline clockwise (TrueType).
01147 };
01148
01149 /**
01150  @brief Build a glyph outline.
01151
01152  This function builds a glyph outline from a Unifont glyph bitmap.
01153
01154  @param[out] result The resulting glyph outline.
01155  @param[in]  bitmap A bitmap array.
01156  @param[in]  byteCount the number of bytes in the input bitmap array.
01157  @param[in]  fillSide Enumerated indicator to fill left or right side.
01158 */
01159 void
01160 buildOutline (Buffer *result, const byte bitmap[], const size_t byteCount,
01161              const enum FillSide fillSide)
01162 {
01163     enum Direction {RIGHT, LEFT, DOWN, UP}; // order is significant
01164
01165     // respective coordinate deltas
01166     const pixels_t dx[] = {1, -1, 0, 0}, dy[] = {0, 0, -1, 1};
01167
01168     assert (byteCount % GLYPH_HEIGHT == 0);
01169     const uint_fast8_t bytesPerRow = byteCount / GLYPH_HEIGHT;
01170     const pixels_t glyphWidth = bytesPerRow * 8;
01171     assert (glyphWidth <= GLYPH_MAX_WIDTH);
01172
01173     #if GLYPH_MAX_WIDTH < 32
01174         typedef uint_fast32_t row_t;
01175     #elif GLYPH_MAX_WIDTH < 64
01176         typedef uint_fast64_t row_t;
01177     #else
01178         #error GLYPH_MAX_WIDTH is too large.
01179     #endif
01180
01181     row_t pixels[GLYPH_HEIGHT + 2] = {0};
01182     for (pixels_t row = GLYPH_HEIGHT; row > 0; row--)
01183         for (pixels_t b = 0; b < bytesPerRow; b++)
01184             pixels[row] = pixels[row] « 8 | *bitmap++;
01185     typedef row_t graph_t[GLYPH_HEIGHT + 1];
01186     graph_t vectors[4];
01187     const row_t *lower = pixels, *upper = pixels + 1;
01188     for (pixels_t row = 0; row <= GLYPH_HEIGHT; row++)
01189     {
01190         const row_t m = (fillSide == FILL_RIGHT) - 1;
01191         vectors[RIGHT][row] = (m ^ (*lower « 1)) & (~m ^ (*upper « 1));
01192         vectors[LEFT][row] = (m ^ (*upper )) & (~m ^ (*lower ));
01193         vectors[DOWN][row] = (m ^ (*lower )) & (~m ^ (*lower « 1));
01194         vectors[UP][row] = (m ^ (*upper « 1)) & (~m ^ (*upper ));
01195         lower++;
01196         upper++;
01197     }
01198     graph_t selection = {0};
01199     const row_t x0 = (row_t)1 « glyphWidth;
01200
01201     /// Get the value of a given bit that is in a given row.
01202     #define getRowBit(rows, x, y) ((rows)[(y)] & x0 » (x))
01203
01204     /// Invert the value of a given bit that is in a given row.
01205     #define flipRowBit(rows, x, y) ((rows)[(y)] ^ x0 » (x))
01206
01207     for (pixels_t y = GLYPH_HEIGHT; y >= 0; y--)
01208     {
01209         for (pixels_t x = 0; x <= glyphWidth; x++)
01210         {

```

```

01211     assert (!getRowBit (vectors[LEFT], x, y));
01212     assert (!getRowBit (vectors[UP], x, y));
01213     enum Direction initial;
01214
01215     if (getRowBit (vectors[RIGHT], x, y))
01216         initial = RIGHT;
01217     else if (getRowBit (vectors[DOWN], x, y))
01218         initial = DOWN;
01219     else
01220         continue;
01221
01222     static_assert ((GLYPH_MAX_WIDTH + 1) * (GLYPH_HEIGHT + 1) * 2 <=
01223         U16MAX, "potential overflow");
01224
01225     uint_fast16_t lastPointCount = 0;
01226     for (bool converged = false;;)
01227     {
01228         uint_fast16_t pointCount = 0;
01229         enum Direction heading = initial;
01230         for (pixels_t tx = x, ty = y;;)
01231         {
01232             if (converged)
01233             {
01234                 storePixels (result, OP_POINT);
01235                 storePixels (result, tx);
01236                 storePixels (result, ty);
01237             }
01238             do
01239             {
01240                 if (converged)
01241                     flipRowBit (vectors[heading], tx, ty);
01242                 tx += dx[heading];
01243                 ty += dy[heading];
01244             } while (getRowBit (vectors[heading], tx, ty));
01245             if (tx == x && ty == y)
01246                 break;
01247             static_assert ((UP ^ DOWN) == 1 && (LEFT ^ RIGHT) == 1,
01248                 "wrong enums");
01249             heading = (heading & 2) ^ 2;
01250             heading |= !getRowBit (selection, tx, ty);
01251             heading ^= !getRowBit (vectors[heading], tx, ty);
01252             assert (getRowBit (vectors[heading], tx, ty));
01253             flipRowBit (selection, tx, ty);
01254             pointCount++;
01255         }
01256         if (converged)
01257             break;
01258         converged = pointCount == lastPointCount;
01259         lastPointCount = pointCount;
01260     }
01261     storePixels (result, OP_CLOSE);
01262 }
01263 }
01264 }
01265 #undef getRowBit
01266 #undef flipRowBit
01267 }
01268
01269 /**
01270  @brief Prepare 32-bit glyph offsets in a font table.
01271
01272  @param[in] sizes Array of glyph sizes, for offset calculations.
01273  */
01274 void
01275 prepareOffsets (size_t *sizes)
01276 {
01277     size_t *p = sizes;
01278     for (size_t *i = sizes + 1; *i; i++)
01279         *i += *p++;
01280     if (*p > 2147483647U) // offset not representable
01281         fail ("CFF table is too large.");
01282 }
01283
01284 /**
01285  @brief Prepare a font name string index.
01286
01287  @param[in] names List of name strings.
01288  @return Pointer to a Buffer struct containing the string names.
01289  */
01290 Buffer *
01291 prepareStringIndex (const NameStrings names)

```

```

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 > 0xffffffff);
01310     cacheU16 (buf, stringCount); // count
01311     cacheU8 (buf, offsetSize); // offsetSize
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 }
01320
01321 /**
01322     @brief Add a CFF table to a font.
01323
01324     @param[in,out] font Pointer to a Font struct to contain the CFF table.
01325     @param[in] version Version of CFF table, with value 1 or 2.
01326     @param[in] names List of NameStrings.
01327 */
01328 void
01329 fillCFF (Font *font, int version, const NameStrings names)
01330 {
01331     // HACK: For convenience, CFF data structures are hard coded.
01332     assert (0 < version && version <= 2);
01333     Buffer *cff = newBuffer (65536);
01334     addTable (font, version == 1 ? "CFF " : "CFF2", cff);
01335
01336     /// Use fixed width integer for variables to simplify offset calculation.
01337     #define cacheCFF32(buf, x) (cacheU8 ((buf), 29), cacheU32 ((buf), (x)))
01338
01339     // In Unifont, 16px glyphs are more common. This is used by CFF1 only.
01340     const pixels_t defaultWidth = 16, nominalWidth = 8;
01341     if (version == 1)
01342     {
01343         Buffer *strings = prepareStringIndex (names);
01344         size_t stringsSize = countBufferedBytes (strings);
01345         const char *cffName = names[6];
01346         assert (cffName);
01347         size_t nameLength = strlen (cffName);
01348         size_t namesSize = nameLength + 5;
01349         // These sizes must be updated together with the data below.
01350         size_t offsets[] = {4, namesSize, 45, stringsSize, 2, 5, 8, 32, 4, 0};
01351         prepareOffsets (offsets);
01352         { // Header
01353             cacheU8 (cff, 1); // major
01354             cacheU8 (cff, 0); // minor
01355             cacheU8 (cff, 4); // hdrSize
01356             cacheU8 (cff, 1); // offsetSize
01357         }
01358         assert (countBufferedBytes (cff) == offsets[0]);
01359         { // Name INDEX (should not be used by OpenType readers)
01360             cacheU16 (cff, 1); // count
01361             cacheU8 (cff, 1); // offsetSize
01362             cacheU8 (cff, 1); // offset[0]
01363             if (nameLength + 1 > 255) // must be too long; spec limit is 63
01364                 fail ("PostScript name is too long.");
01365             cacheU8 (cff, nameLength + 1); // offset[1]
01366             cacheBytes (cff, cffName, nameLength);
01367         }
01368         assert (countBufferedBytes (cff) == offsets[1]);
01369         { // Top DICT INDEX
01370             cacheU16 (cff, 1); // count
01371             cacheU8 (cff, 1); // offsetSize
01372             cacheU8 (cff, 1); // offset[0]

```

```

01373     cacheU8 (cff, 41); // offset[1]
01374     cacheCFFOperand (cff, 391); // "Adobe"
01375     cacheCFFOperand (cff, 392); // "Identity"
01376     cacheCFFOperand (cff, 0);
01377     cacheBytes (cff, (byte[]){12, 30}, 2); // ROS
01378     cacheCFF32 (cff, font->glyphCount);
01379     cacheBytes (cff, (byte[]){12, 34}, 2); // CIDCount
01380     cacheCFF32 (cff, offsets[6]);
01381     cacheBytes (cff, (byte[]){12, 36}, 2); // FDArray
01382     cacheCFF32 (cff, offsets[5]);
01383     cacheBytes (cff, (byte[]){12, 37}, 2); // FDSelect
01384     cacheCFF32 (cff, offsets[4]);
01385     cacheU8 (cff, 15); // charset
01386     cacheCFF32 (cff, offsets[8]);
01387     cacheU8 (cff, 17); // CharStrings
01388 }
01389 assert (countBufferedBytes (cff) == offsets[2]);
01390 { // String INDEX
01391     cacheBuffer (cff, strings);
01392     freeBuffer (strings);
01393 }
01394 assert (countBufferedBytes (cff) == offsets[3]);
01395 cacheU16 (cff, 0); // Global Subr INDEX
01396 assert (countBufferedBytes (cff) == offsets[4]);
01397 { // Charsets
01398     cacheU8 (cff, 2); // format
01399     { // Range2[0]
01400         cacheU16 (cff, 1); // first
01401         cacheU16 (cff, font->glyphCount - 2); // nLeft
01402     }
01403 }
01404 assert (countBufferedBytes (cff) == offsets[5]);
01405 { // FDSelect
01406     cacheU8 (cff, 3); // format
01407     cacheU16 (cff, 1); // nRanges
01408     cacheU16 (cff, 0); // first
01409     cacheU8 (cff, 0); // fd
01410     cacheU16 (cff, font->glyphCount); // sentinel
01411 }
01412 assert (countBufferedBytes (cff) == offsets[6]);
01413 { // FDArray
01414     cacheU16 (cff, 1); // count
01415     cacheU8 (cff, 1); // offSize
01416     cacheU8 (cff, 1); // offset[0]
01417     cacheU8 (cff, 28); // offset[1]
01418     cacheCFFOperand (cff, 393);
01419     cacheBytes (cff, (byte[]){12, 38}, 2); // FontName
01420     // Windows requires FontMatrix in Font DICT.
01421     const byte unit[] = {0x1e, 0x15, 0x62, 0x5c, 0x6f}; // 1/64 (0.015625)
01422     cacheBytes (cff, unit, sizeof unit);
01423     cacheCFFOperand (cff, 0);
01424     cacheCFFOperand (cff, 0);
01425     cacheBytes (cff, unit, sizeof unit);
01426     cacheCFFOperand (cff, 0);
01427     cacheCFFOperand (cff, 0);
01428     cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
01429     cacheCFFOperand (cff, offsets[8] - offsets[7]); // size
01430     cacheCFF32 (cff, offsets[7]); // offset
01431     cacheU8 (cff, 18); // Private
01432 }
01433 assert (countBufferedBytes (cff) == offsets[7]);
01434 { // Private
01435     cacheCFFOperand (cff, FU (defaultWidth));
01436     cacheU8 (cff, 20); // defaultWidthX
01437     cacheCFFOperand (cff, FU (nominalWidth));
01438     cacheU8 (cff, 21); // nominalWidthX
01439 }
01440 assert (countBufferedBytes (cff) == offsets[8]);
01441 }
01442 else
01443 {
01444     assert (version == 2);
01445     // These sizes must be updated together with the data below.
01446     size_t offsets[] = {5, 21, 4, 10, 0};
01447     prepareOffsets (offsets);
01448     { // Header
01449         cacheU8 (cff, 2); // majorVersion
01450         cacheU8 (cff, 0); // minorVersion
01451         cacheU8 (cff, 5); // headerSize
01452         cacheU16 (cff, offsets[1] - offsets[0]); // topDictLength
01453     }

```

```

01454     assert (countBufferedBytes (cff) == offsets[0]);
01455     { // Top DICT
01456         const byte unit[] = {0x1e,0x15,0x62,0x5c,0x6f}; // 1/64 (0.015625)
01457         cacheBytes (cff, unit, sizeof unit);
01458         cacheCFFOperand (cff, 0);
01459         cacheCFFOperand (cff, 0);
01460         cacheBytes (cff, unit, sizeof unit);
01461         cacheCFFOperand (cff, 0);
01462         cacheCFFOperand (cff, 0);
01463         cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
01464         cacheCFFOperand (cff, offsets[2]);
01465         cacheBytes (cff, (byte[]){12, 36}, 2); // FDArray
01466         cacheCFFOperand (cff, offsets[3]);
01467         cacheU8 (cff, 17); // CharStrings
01468     }
01469     assert (countBufferedBytes (cff) == offsets[1]);
01470     cacheU32 (cff, 0); // Global Subr INDEX
01471     assert (countBufferedBytes (cff) == offsets[2]);
01472     { // Font DICT INDEX
01473         cacheU32 (cff, 1); // count
01474         cacheU8 (cff, 1); // offSize
01475         cacheU8 (cff, 1); // offset[0]
01476         cacheU8 (cff, 4); // offset[1]
01477         cacheCFFOperand (cff, 0);
01478         cacheCFFOperand (cff, 0);
01479         cacheU8 (cff, 18); // Private
01480     }
01481     assert (countBufferedBytes (cff) == offsets[3]);
01482 }
01483 { // CharStrings INDEX
01484     Buffer *offsets = newBuffer (4096);
01485     Buffer *charstrings = newBuffer (4096);
01486     Buffer *outline = newBuffer (1024);
01487     const Glyph *glyph = getBufferHead (font->glyphs);
01488     const Glyph *const endGlyph = glyph + font->glyphCount;
01489     for (; glyph < endGlyph; glyph++)
01490     {
01491         // CFF offsets start at 1
01492         storeU32 (offsets, countBufferedBytes (charstrings) + 1);
01493
01494         pixels_t rx = -glyph->pos;
01495         pixels_t ry = DESCENDER;
01496         resetBuffer (outline);
01497         buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_LEFT);
01498         enum CFFOp {rmoveto=21, hmoveto=22, vmoveto=4, hlineto=6,
01499             vlineto=7, endchar=14};
01500         enum CFFOp pendingOp = 0;
01501         const int STACK_LIMIT = version == 1 ? 48 : 513;
01502         int stackSize = 0;
01503         bool isDrawing = false;
01504         pixels_t width = glyph->combining ? 0 : PW (glyph->byteCount);
01505         if (version == 1 && width != defaultWidth)
01506         {
01507             cacheCFFOperand (charstrings, FU (width - nominalWidth));
01508             stackSize++;
01509         }
01510         for (const pixels_t *p = getBufferHead (outline),
01511             *const end = getBufferTail (outline); p < end;)
01512         {
01513             int s = 0;
01514             const enum ContourOp op = *p++;
01515             if (op == OP_POINT)
01516             {
01517                 const pixels_t x = *p++, y = *p++;
01518                 if (x != rx)
01519                 {
01520                     cacheCFFOperand (charstrings, FU (x - rx));
01521                     rx = x;
01522                     stackSize++;
01523                     s |= 1;
01524                 }
01525                 if (y != ry)
01526                 {
01527                     cacheCFFOperand (charstrings, FU (y - ry));
01528                     ry = y;
01529                     stackSize++;
01530                     s |= 2;
01531                 }
01532                 assert (!(isDrawing && s == 3));
01533             }
01534             if (s)

```

```

01535     {
01536         if (lisDrawing)
01537         {
01538             const enum CFFOp moves[] = {0, hmoveto, vmoveto,
01539                 rmoveto};
01540             cacheU8 (charstrings, moves[s]);
01541             stackSize = 0;
01542         }
01543         else if (!pendingOp)
01544             pendingOp = (enum CFFOp[]){0, hlineto, vlineto}[s];
01545     }
01546     else if (!lisDrawing)
01547     {
01548         // only when the first point happens to be (0, 0)
01549         cacheCFFOperand (charstrings, FU (0));
01550         cacheU8 (charstrings, hmoveto);
01551         stackSize = 0;
01552     }
01553     if (op == OP_CLOSE || stackSize >= STACK_LIMIT)
01554     {
01555         assert (stackSize <= STACK_LIMIT);
01556         cacheU8 (charstrings, pendingOp);
01557         pendingOp = 0;
01558         stackSize = 0;
01559     }
01560     isDrawing = op != OP_CLOSE;
01561 }
01562 if (version == 1)
01563     cacheU8 (charstrings, endchar);
01564 }
01565 size_t lastOffset = countBufferedBytes (charstrings) + 1;
01566 #if SIZE_MAX > U32MAX
01567     if (lastOffset > U32MAX)
01568         fail ("CFF data exceeded size limit.");
01569 #endif
01570 storeU32 (offsets, lastOffset);
01571 int offsetSize = 1 + (lastOffset > 0xff)
01572     + (lastOffset > 0xffff)
01573     + (lastOffset > 0xfffff);
01574 // count (must match 'numGlyphs' in 'maxp' table)
01575 cacheU (cff, font->glyphCount, version * 2);
01576 cacheU8 (cff, offsetSize); // offSize
01577 const uint_least32_t *p = getBufferHead (offsets);
01578 const uint_least32_t *const end = getBufferTail (offsets);
01579 for (; p < end; p++)
01580     cacheU (cff, *p, offsetSize); // offsets
01581 cacheBuffer (cff, charstrings); // data
01582 freeBuffer (offsets);
01583 freeBuffer (charstrings);
01584 freeBuffer (outline);
01585 }
01586 #undef cacheCFF32
01587 }
01588
01589 /**
01590  @brief Add a TrueType table to a font.
01591
01592  @param[in,out] font Pointer to a Font struct to contain the TrueType table.
01593  @param[in] format The TrueType "loca" table format, Offset16 or Offset32.
01594  @param[in] names List of NameStrings.
01595 */
01596 void
01597 fillTrueType (Font *font, enum LocaFormat *format,
01598     uint_fast16_t *maxPoints, uint_fast16_t *maxContours)
01599 {
01600     Buffer *glyf = newBuffer (65536);
01601     addTable (font, "glyf", glyf);
01602     Buffer *loca = newBuffer (4 * (font->glyphCount + 1));
01603     addTable (font, "loca", loca);
01604     *format = LOCA_OFFSET32;
01605     Buffer *endPoints = newBuffer (256);
01606     Buffer *flags = newBuffer (256);
01607     Buffer *xs = newBuffer (256);
01608     Buffer *ys = newBuffer (256);
01609     Buffer *outline = newBuffer (1024);
01610     Glyph *const glyphs = getBufferHead (font->glyphs);
01611     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
01612     for (Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
01613     {
01614         cacheU32 (loca, countBufferedBytes (glyf));
01615         pixels_t rx = -glyph->pos;

```

```

01616     pixels_t ry = DESCENDER;
01617     pixels_t xMin = GLYPH_MAX_WIDTH, xMax = 0;
01618     pixels_t yMin = ASCENDER, yMax = -DESCENDER;
01619     resetBuffer (endPoints);
01620     resetBuffer (flags);
01621     resetBuffer (xs);
01622     resetBuffer (ys);
01623     resetBuffer (outline);
01624     buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_RIGHT);
01625     uint_fast32_t pointCount = 0, contourCount = 0;
01626     for (const pixels_t *p = getBufferHead (outline),
01627          *const end = getBufferTail (outline); p < end;)
01628     {
01629         const enum ContourOp op = *p++;
01630         if (op == OP_CLOSE)
01631         {
01632             contourCount++;
01633             assert (contourCount <= U16MAX);
01634             cacheU16 (endPoints, pointCount - 1);
01635             continue;
01636         }
01637         assert (op == OP_POINT);
01638         pointCount++;
01639         assert (pointCount <= U16MAX);
01640         const pixels_t x = *p++, y = *p++;
01641         uint_fast8_t pointFlags =
01642             + B1 (0) // point is on curve
01643             + BX (1, x != rx) // x coordinate is 1 byte instead of 2
01644             + BX (2, y != ry) // y coordinate is 1 byte instead of 2
01645             + B0 (3) // repeat
01646             + BX (4, x >= rx) // when x is 1 byte: x is positive;
01647               // when x is 2 bytes: x unchanged and omitted
01648             + BX (5, y >= ry) // when y is 1 byte: y is positive;
01649               // when y is 2 bytes: y unchanged and omitted
01650             + B1 (6) // contours may overlap
01651             + B0 (7) // reserved
01652         ;
01653         cacheU8 (flags, pointFlags);
01654         if (x != rx)
01655             cacheU8 (xs, FU (x > rx ? x - rx : rx - x));
01656         if (y != ry)
01657             cacheU8 (ys, FU (y > ry ? y - ry : ry - y));
01658         if (x < xMin) xMin = x;
01659         if (y < yMin) yMin = y;
01660         if (x > xMax) xMax = x;
01661         if (y > yMax) yMax = y;
01662         rx = x;
01663         ry = y;
01664     }
01665     if (contourCount == 0)
01666         continue; // blank glyph is indicated by the 'loca' table
01667     glyph->lsb = glyph->pos + xMin;
01668     cacheU16 (glyf, contourCount); // numberOfContours
01669     cacheU16 (glyf, FU (glyph->pos + xMin)); // xMin
01670     cacheU16 (glyf, FU (yMin)); // yMin
01671     cacheU16 (glyf, FU (glyph->pos + xMax)); // xMax
01672     cacheU16 (glyf, FU (yMax)); // yMax
01673     cacheBuffer (glyf, endPoints); // endPtsOfContours[]
01674     cacheU16 (glyf, 0); // instructionLength
01675     cacheBuffer (glyf, flags); // flags[]
01676     cacheBuffer (glyf, xs); // xCoordinates[]
01677     cacheBuffer (glyf, ys); // yCoordinates[]
01678     if (pointCount > *maxPoints)
01679         *maxPoints = pointCount;
01680     if (contourCount > *maxContours)
01681         *maxContours = contourCount;
01682 }
01683 cacheU32 (loca, countBufferedBytes (glyf));
01684 freeBuffer (endPoints);
01685 freeBuffer (flags);
01686 freeBuffer (xs);
01687 freeBuffer (ys);
01688 freeBuffer (outline);
01689 }
01690
01691 /**
01692  @brief Create a dummy blank outline in a font table.
01693
01694  @param[in,out] font Pointer to a Font struct to insert a blank outline.
01695 */
01696 void

```

```

01697 fillBlankOutline (Font *font)
01698 {
01699     Buffer *glyf = newBuffer (12);
01700     addTable (font, "glyf", glyf);
01701     // Empty table is not allowed, but an empty outline for glyph 0 suffices.
01702     cacheU16 (glyf, 0); // numberOfContours
01703     cacheU16 (glyf, FU (0)); // xMin
01704     cacheU16 (glyf, FU (0)); // yMin
01705     cacheU16 (glyf, FU (0)); // xMax
01706     cacheU16 (glyf, FU (0)); // yMax
01707     cacheU16 (glyf, 0); // instructionLength
01708     Buffer *loca = newBuffer (2 * (font->glyphCount + 1));
01709     addTable (font, "loca", loca);
01710     cacheU16 (loca, 0); // offsets[0]
01711     assert (countBufferedBytes (glyf) % 2 == 0);
01712     for (uint_fast32_t i = 1; i <= font->glyphCount; i++)
01713         cacheU16 (loca, countBufferedBytes (glyf) / 2); // offsets[i]
01714 }
01715
01716 /**
01717  * @brief Fill OpenType bitmap data and location tables.
01718  *
01719  * This function fills an Embedded Bitmap Data (EBDT) Table
01720  * and an Embedded Bitmap Location (EBLC) Table with glyph
01721  * bitmap information. These tables enable embedding bitmaps
01722  * in OpenType fonts. No Embedded Bitmap Scaling (EBS) table
01723  * is used for the bitmap glyphs, only EBDT and EBLC.
01724  *
01725  * @param[in,out] font Pointer to a Font struct in which to add bitmaps.
01726  */
01727 void
01728 fillBitmap (Font *font)
01729 {
01730     const Glyph *const glyphs = getBufferHead (font->glyphs);
01731     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
01732     size_t bitmapsSize = 0;
01733     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
01734         bitmapsSize += glyph->byteCount;
01735     Buffer *ebdt = newBuffer (4 + bitmapsSize);
01736     addTable (font, "EBDT", ebdt);
01737     cacheU16 (ebdt, 2); // majorVersion
01738     cacheU16 (ebdt, 0); // minorVersion
01739     uint_fast8_t byteCount = 0; // unequal to any glyph
01740     pixels_t pos = 0;
01741     bool combining = false;
01742     Buffer *rangeHeads = newBuffer (32);
01743     Buffer *offsets = newBuffer (64);
01744     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
01745     {
01746         if (glyph->byteCount != byteCount || glyph->pos != pos ||
01747             glyph->combining != combining)
01748         {
01749             storeU16 (rangeHeads, glyph - glyphs);
01750             storeU32 (offsets, countBufferedBytes (ebdt));
01751             byteCount = glyph->byteCount;
01752             pos = glyph->pos;
01753             combining = glyph->combining;
01754         }
01755         cacheBytes (ebdt, glyph->bitmap, byteCount);
01756     }
01757     const uint_least16_t *ranges = getBufferHead (rangeHeads);
01758     const uint_least16_t *rangesEnd = getBufferTail (rangeHeads);
01759     uint_fast32_t rangeCount = rangesEnd - ranges;
01760     storeU16 (rangeHeads, font->glyphCount);
01761     Buffer *eblc = newBuffer (4096);
01762     addTable (font, "EBLC", eblc);
01763     cacheU16 (eblc, 2); // majorVersion
01764     cacheU16 (eblc, 0); // minorVersion
01765     cacheU32 (eblc, 1); // numSizes
01766     { // bitmapSizes[0]
01767         cacheU32 (eblc, 56); // indexSubTableArrayOffset
01768         cacheU32 (eblc, (8 + 20) * rangeCount); // indexTablesSize
01769         cacheU32 (eblc, rangeCount); // numberOfIndexSubTables
01770         cacheU32 (eblc, 0); // colorRef
01771         { // hori
01772             cacheU8 (eblc, ASCENDER); // ascender
01773             cacheU8 (eblc, -DESCENDER); // descender
01774             cacheU8 (eblc, font->maxWidth); // widthMax
01775             cacheU8 (eblc, 1); // caretSlopeNumerator
01776             cacheU8 (eblc, 0); // caretSlopeDenominator
01777             cacheU8 (eblc, 0); // caretOffset

```



```

01778     cacheU8 (eblc, 0); // minOriginSB
01779     cacheU8 (eblc, 0); // minAdvanceSB
01780     cacheU8 (eblc, ASCENDER); // maxBeforeBL
01781     cacheU8 (eblc, -DESCENDER); // minAfterBL
01782     cacheU8 (eblc, 0); // pad1
01783     cacheU8 (eblc, 0); // pad2
01784 }
01785 { // vert
01786     cacheU8 (eblc, ASCENDER); // ascender
01787     cacheU8 (eblc, -DESCENDER); // descender
01788     cacheU8 (eblc, font->maxWidth); // widthMax
01789     cacheU8 (eblc, 1); // caretSlopeNumerator
01790     cacheU8 (eblc, 0); // caretSlopeDenominator
01791     cacheU8 (eblc, 0); // caretOffset
01792     cacheU8 (eblc, 0); // minOriginSB
01793     cacheU8 (eblc, 0); // minAdvanceSB
01794     cacheU8 (eblc, ASCENDER); // maxBeforeBL
01795     cacheU8 (eblc, -DESCENDER); // minAfterBL
01796     cacheU8 (eblc, 0); // pad1
01797     cacheU8 (eblc, 0); // pad2
01798 }
01799 cacheU16 (eblc, 0); // startGlyphIndex
01800 cacheU16 (eblc, font->glyphCount - 1); // endGlyphIndex
01801 cacheU8 (eblc, 16); // ppemX
01802 cacheU8 (eblc, 16); // ppemY
01803 cacheU8 (eblc, 1); // bitDepth
01804 cacheU8 (eblc, 1); // flags = Horizontal
01805 }
01806 { // IndexSubTableArray
01807     uint_fast32_t offset = rangeCount * 8;
01808     for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
01809     {
01810         cacheU16 (eblc, *p); // firstGlyphIndex
01811         cacheU16 (eblc, p[1] - 1); // lastGlyphIndex
01812         cacheU32 (eblc, offset); // additionalOffsetToIndexSubtable
01813         offset += 20;
01814     }
01815 }
01816 { // IndexSubTables
01817     const uint_least32_t *offset = getBufferHead (offsets);
01818     for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
01819     {
01820         const Glyph *glyph = &glyphs[*p];
01821         cacheU16 (eblc, 2); // indexFormat
01822         cacheU16 (eblc, 5); // imageFormat
01823         cacheU32 (eblc, *offset++); // imageDataOffset
01824         cacheU32 (eblc, glyph->byteCount); // imageSize
01825         { // bigMetrics
01826             cacheU8 (eblc, GLYPH_HEIGHT); // height
01827             const uint_fast8_t width = PW (glyph->byteCount);
01828             cacheU8 (eblc, width); // width
01829             cacheU8 (eblc, glyph->pos); // horiBearingX
01830             cacheU8 (eblc, ASCENDER); // horiBearingY
01831             cacheU8 (eblc, glyph->combining ? 0 : width); // horiAdvance
01832             cacheU8 (eblc, 0); // vertBearingX
01833             cacheU8 (eblc, 0); // vertBearingY
01834             cacheU8 (eblc, GLYPH_HEIGHT); // vertAdvance
01835         }
01836     }
01837 }
01838 freeBuffer (rangeHeads);
01839 freeBuffer (offsets);
01840 }
01841
01842 /**
01843  @brief Fill a "head" font table.
01844
01845  The "head" table contains font header information common to the
01846  whole font.
01847
01848  @param[in,out] font The Font struct to which to add the table.
01849  @param[in] locaFormat The "loca" offset index location table.
01850  @param[in] xMin The minimum x-coordinate for a glyph.
01851 */
01852 void
01853 fillHeadTable (Font *font, enum LocaFormat locaFormat, pixels_t xMin)
01854 {
01855     Buffer *head = newBuffer (56);
01856     addTable (font, "head", head);
01857     cacheU16 (head, 1); // majorVersion
01858     cacheU16 (head, 0); // minorVersion

```

```

01859 cacheZeros (head, 4); // fontRevision (unused)
01860 // The 'checksumAdjustment' field is a checksum of the entire file.
01861 // It is later calculated and written directly in the 'writeFont' function.
01862 cacheU32 (head, 0); // checksumAdjustment (placeholder)
01863 cacheU32 (head, 0x5f0f3cf5); // magicNumber
01864 const uint_fast16_t flags =
01865     + B1 (0) // baseline at y=0
01866     + B1 (1) // LSB at x=0 (doubtful; probably should be LSB=xMin)
01867     + B0 (2) // instructions may depend on point size
01868     + B0 (3) // force internal ppem to integers
01869     + B0 (4) // instructions may alter advance width
01870     + B0 (5) // not used in OpenType
01871     + B0 (6) // not used in OpenType
01872     + B0 (7) // not used in OpenType
01873     + B0 (8) // not used in OpenType
01874     + B0 (9) // not used in OpenType
01875     + B0 (10) // not used in OpenType
01876     + B0 (11) // font transformed
01877     + B0 (12) // font converted
01878     + B0 (13) // font optimized for ClearType
01879     + B0 (14) // last resort font
01880     + B0 (15) // reserved
01881 ;
01882 cacheU16 (head, flags); // flags
01883 cacheU16 (head, FUPEM); // unitsPerEm
01884 cacheZeros (head, 8); // created (unused)
01885 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 }
01907
01908 /**
01909  @brief Fill a "hhea" font table.
01910
01911  The "hhea" table contains horizontal header information,
01912  for example left and right side bearings.
01913
01914  @param[in,out] font The Font struct to which to add the table.
01915  @param[in] xMin The minimum x-coordinate for a glyph.
01916 */
01917 void
01918 fillHheaTable (Font *font, pixels_t xMin)
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); // numberOfMetrics

```

```

01940 }
01941
01942 /**
01943  @brief Fill a "maxp" font table.
01944
01945  The "maxp" table contains maximum profile information,
01946  such as the memory required to contain the font.
01947
01948  @param[in,out] font The Font struct to which to add the table.
01949  @param[in] isCFF true if a CFF font is included, false otherwise.
01950  @param[in] maxPoints Maximum points in a non-composite glyph.
01951  @param[in] maxContours Maximum contours in a non-composite glyph.
01952 */
01953 void
01954 fillMaxpTable (Font *font, bool isCFF, uint_fast16_t maxPoints,
01955               uint_fast16_t maxContours)
01956 {
01957     Buffer *maxp = newBuffer (32);
01958     addTable (font, "maxp", maxp);
01959     cacheU32 (maxp, isCFF ? 0x00005000 : 0x00010000); // version
01960     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 }
01977
01978 /**
01979  @brief Fill an "OS/2" font table.
01980
01981  The "OS/2" table contains OS/2 and Windows font metrics information.
01982
01983  @param[in,out] font The Font struct to which to add the table.
01984 */
01985 void
01986 fillOS2Table (Font *font)
01987 {
01988     Buffer *os2 = newBuffer (100);
01989     addTable (font, "OS/2", os2);
01990     cacheU16 (os2, 5); // version
01991     // HACK: Average glyph width is not actually calculated.
01992     cacheU16 (os2, FU (font->maxWidth)); // xAvgCharWidth
01993     cacheU16 (os2, 400); // usWeightClass = Normal
01994     cacheU16 (os2, 5); // usWidthClass = Medium
01995     const uint_fast16_t typeFlags =
01996         + B0 (0) // reserved
01997         // usage permissions, one of:
01998         // Default: Installable embedding
01999         + B0 (1) // Restricted License embedding
02000         + B0 (2) // Preview & Print embedding
02001         + B0 (3) // Editable embedding
02002         // 4-7 reserved
02003         + B0 (8) // no subsetting
02004         + B0 (9) // bitmap embedding only
02005         // 10-15 reserved
02006     ;
02007     cacheU16 (os2, typeFlags); // fsType
02008     cacheU16 (os2, FU (5)); // ySubscriptXSize
02009     cacheU16 (os2, FU (7)); // ySubscriptYSize
02010     cacheU16 (os2, FU (0)); // ySubscriptXOffset
02011     cacheU16 (os2, FU (1)); // ySubscriptYOffset
02012     cacheU16 (os2, FU (5)); // ySuperscriptXSize
02013     cacheU16 (os2, FU (7)); // ySuperscriptYSize
02014     cacheU16 (os2, FU (0)); // ySuperscriptXOffset
02015     cacheU16 (os2, FU (4)); // ySuperscriptYOffset
02016     cacheU16 (os2, FU (1)); // yStrikeoutSize
02017     cacheU16 (os2, FU (5)); // yStrikeoutPosition
02018     cacheU16 (os2, 0x080a); // sFamilyClass = Sans Serif, Matrix
02019     const byte panose[] =
02020     {

```

```

02021     2, // Family Kind = Latin Text
02022     11, // Serif Style = Normal Sans
02023     4, // Weight = Thin
02024     // Windows would render all glyphs to the same width,
02025     // if 'Proportion' is set to 'Monospaced' (as Unifont should be).
02026     // 'Condensed' is the best alternative according to metrics.
02027     6, // Proportion = Condensed
02028     2, // Contrast = None
02029     2, // Stroke = No Variation
02030     2, // Arm Style = Straight Arms
02031     8, // Letterform = Normal/Square
02032     2, // Midline = Standard/Trimmed
02033     4, // X-height = Constant/Large
02034 };
02035 cacheBytes (os2, panose, sizeof panose); // panose
02036 // HACK: All defined Unicode ranges are marked functional for convenience.
02037 cacheU32 (os2, 0xffffffff); // ulUnicodeRange1
02038 cacheU32 (os2, 0xffffffff); // ulUnicodeRange2
02039 cacheU32 (os2, 0xffffffff); // ulUnicodeRange3
02040 cacheU32 (os2, 0x0effffff); // ulUnicodeRange4
02041 cacheBytes (os2, "GNU ", 4); // achVendID
02042 // fsSelection (must agree with 'macStyle' in 'head' table)
02043 const uint_fast16_t selection =
02044     + B0 (0) // italic
02045     + B0 (1) // underscored
02046     + B0 (2) // negative
02047     + B0 (3) // outlined
02048     + B0 (4) // strikeout
02049     + B0 (5) // bold
02050     + B1 (6) // regular
02051     + B1 (7) // use sTypo* metrics in this table
02052     + B1 (8) // font name conforms to WWS model
02053     + B0 (9) // oblique
02054     // 10-15 reserved
02055 ;
02056 cacheU16 (os2, selection);
02057 const Glyph *glyphs = getBufferHead (font->glyphs);
02058 uint_fast32_t first = glyphs[1].codePoint;
02059 uint_fast32_t last = glyphs[font->glyphCount - 1].codePoint;
02060 cacheU16 (os2, first < U16MAX ? first : U16MAX); // usFirstCharIndex
02061 cacheU16 (os2, last < U16MAX ? last : U16MAX); // usLastCharIndex
02062 cacheU16 (os2, FU (ASCENDER)); // sTypoAscender
02063 cacheU16 (os2, FU (-DESCENDER)); // sTypoDescender
02064 cacheU16 (os2, FU (0)); // sTypoLineGap
02065 cacheU16 (os2, FU (ASCENDER)); // usWinAscent
02066 cacheU16 (os2, FU (DESCENDER)); // usWinDescent
02067 // HACK: All reasonable code pages are marked functional for convenience.
02068 cacheU32 (os2, 0x603f01ff); // ulCodePageRange1
02069 cacheU32 (os2, 0xffff0000); // ulCodePageRange2
02070 cacheU16 (os2, FU (8)); // sxHeight
02071 cacheU16 (os2, FU (10)); // sCapHeight
02072 cacheU16 (os2, 0); // usDefaultChar
02073 cacheU16 (os2, 0x20); // usBreakChar
02074 cacheU16 (os2, 0); // usMaxContext
02075 cacheU16 (os2, 0); // usLowerOpticalPointSize
02076 cacheU16 (os2, 0xffff); // usUpperOpticalPointSize
02077 }
02078 /**
02079  @brief Fill an "hmtx" font table.
02080
02081  The "hmtx" table contains horizontal metrics information.
02082
02083  @param[in,out] font The Font struct to which to add the table.
02084 */
02085 void
02086 fillHmtxTable (Font *font)
02087 {
02088     Buffer *hmtx = newBuffer (4 * font->glyphCount);
02089     addTable (font, "hmtx", hmtx);
02090     const Glyph *const glyphs = getBufferHead (font->glyphs);
02091     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
02092     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
02093     {
02094         int_fast16_t aw = glyph->combining ? 0 : PW (glyph->byteCount);
02095         cacheU16 (hmtx, FU (aw)); // advanceWidth
02096         cacheU16 (hmtx, FU (glyph->lsb)); // lsb
02097     }
02098 }
02099 }
02100
02101 /**

```

```

02102  @brief Fill a "cmap" font table.
02103
02104  The "cmap" table contains character to glyph index mapping information.
02105
02106  @param[in,out] font The Font struct to which to add the table.
02107 */
02108 void
02109 fillCmapTable (Font *font)
02110 {
02111     Glyph *const glyphs = getBufferHead (font->glyphs);
02112     Buffer *rangeHeads = newBuffer (16);
02113     uint_fast32_t rangeCount = 0;
02114     uint_fast32_t bmpRangeCount = 1; // 1 for the last 0xffff-0xffff range
02115     glyphs[0].codePoint = glyphs[1].codePoint; // to start a range at glyph 1
02116     for (uint_fast16_t i = 1; i < font->glyphCount; i++)
02117     {
02118         if (glyphs[i].codePoint != glyphs[i - 1].codePoint + 1)
02119         {
02120             storeU16 (rangeHeads, i);
02121             rangeCount++;
02122             bmpRangeCount += glyphs[i].codePoint < 0xffff;
02123         }
02124     }
02125     Buffer *cmap = newBuffer (256);
02126     addTable (font, "cmap", cmap);
02127     // Format 4 table is always generated for compatibility.
02128     bool hasFormat12 = glyphs[font->glyphCount - 1].codePoint > 0xffff;
02129     cacheU16 (cmap, 0); // version
02130     cacheU16 (cmap, 1 + hasFormat12); // numTables
02131     { // encodingRecords[0]
02132         cacheU16 (cmap, 3); // platformID
02133         cacheU16 (cmap, 1); // encodingID
02134         cacheU32 (cmap, 12 + 8 * hasFormat12); // subtableOffset
02135     }
02136     if (hasFormat12) // encodingRecords[1]
02137     {
02138         cacheU16 (cmap, 3); // platformID
02139         cacheU16 (cmap, 10); // encodingID
02140         cacheU32 (cmap, 36 + 8 * bmpRangeCount); // subtableOffset
02141     }
02142     const uint_least16_t *ranges = getBufferHead (rangeHeads);
02143     const uint_least16_t *const rangesEnd = getBufferTail (rangeHeads);
02144     storeU16 (rangeHeads, font->glyphCount);
02145     { // format 4 table
02146         cacheU16 (cmap, 4); // format
02147         cacheU16 (cmap, 16 + 8 * bmpRangeCount); // length
02148         cacheU16 (cmap, 0); // language
02149         if (bmpRangeCount * 2 > U16MAX)
02150             fail ("Too many ranges in 'cmap' table.");
02151         cacheU16 (cmap, bmpRangeCount * 2); // segCountX2
02152         uint_fast16_t searchRange = 1, entrySelector = -1;
02153         while (searchRange <= bmpRangeCount)
02154         {
02155             searchRange <<= 1;
02156             entrySelector++;
02157         }
02158         cacheU16 (cmap, searchRange); // searchRange
02159         cacheU16 (cmap, entrySelector); // entrySelector
02160         cacheU16 (cmap, bmpRangeCount * 2 - searchRange); // rangeShift
02161         { // endCode[]
02162             const uint_least16_t *p = ranges;
02163             for (p++; p < rangesEnd && glyphs[*p].codePoint < 0xffff; p++)
02164                 cacheU16 (cmap, glyphs[*p - 1].codePoint);
02165             uint_fast32_t cp = glyphs[*p - 1].codePoint;
02166             if (cp > 0xfffe)
02167                 cp = 0xfffe;
02168             cacheU16 (cmap, cp);
02169             cacheU16 (cmap, 0xffff);
02170         }
02171         cacheU16 (cmap, 0); // reservedPad
02172         { // startCode[]
02173             for (uint_fast32_t i = 0; i < bmpRangeCount - 1; i++)
02174                 cacheU16 (cmap, glyphs[ranges[i]].codePoint);
02175             cacheU16 (cmap, 0xffff);
02176         }
02177         { // idDelta[]
02178             const uint_least16_t *p = ranges;
02179             for (; p < rangesEnd && glyphs[*p].codePoint < 0xffff; p++)
02180                 cacheU16 (cmap, *p - glyphs[*p].codePoint);
02181             uint_fast16_t delta = 1;
02182             if (p < rangesEnd && *p == 0xffff)

```

```

02183         delta = *p - glyphs[*p].codePoint;
02184         cacheU16 (cmap, delta);
02185     }
02186     { // idRangeOffsets[]
02187         for (uint_least16_t i = 0; i < bmpRangeCount; i++)
02188             cacheU16 (cmap, 0);
02189     }
02190 }
02191 if (hasFormat12) // format 12 table
02192 {
02193     cacheU16 (cmap, 12); // format
02194     cacheU16 (cmap, 0); // reserved
02195     cacheU32 (cmap, 16 + 12 * rangeCount); // length
02196     cacheU32 (cmap, 0); // language
02197     cacheU32 (cmap, rangeCount); // numGroups
02198
02199     // groups[]
02200     for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
02201     {
02202         cacheU32 (cmap, glyphs[*p].codePoint); // startCharCode
02203         cacheU32 (cmap, glyphs[p[1] - 1].codePoint); // endCharCode
02204         cacheU32 (cmap, *p); // startGlyphID
02205     }
02206 }
02207 freeBuffer (rangeHeads);
02208 }
02209
02210 /**
02211  * @brief Fill a "post" font table.
02212  *
02213  * The "post" table contains information for PostScript printers.
02214  *
02215  * @param[in,out] font The Font struct to which to add the table.
02216  */
02217 void
02218 fillPostTable (Font *font)
02219 {
02220     Buffer *post = newBuffer (32);
02221     addTable (font, "post", post);
02222     cacheU32 (post, 0x00030000); // version = 3.0
02223     cacheU32 (post, 0); // italicAngle
02224     cacheU16 (post, 0); // underlinePosition
02225     cacheU16 (post, 1); // underlineThickness
02226     cacheU32 (post, 1); // isFixedPitch
02227     cacheU32 (post, 0); // minMemType42
02228     cacheU32 (post, 0); // maxMemType42
02229     cacheU32 (post, 0); // minMemType1
02230     cacheU32 (post, 0); // maxMemType1
02231 }
02232
02233 /**
02234  * @brief Fill a "GPOS" font table.
02235  *
02236  * The "GPOS" table contains information for glyph positioning.
02237  *
02238  * @param[in,out] font The Font struct to which to add the table.
02239  */
02240 void
02241 fillGposTable (Font *font)
02242 {
02243     Buffer *gpos = newBuffer (16);
02244     addTable (font, "GPOS", gpos);
02245     cacheU16 (gpos, 1); // majorVersion
02246     cacheU16 (gpos, 0); // minorVersion
02247     cacheU16 (gpos, 10); // scriptListOffset
02248     cacheU16 (gpos, 12); // featureListOffset
02249     cacheU16 (gpos, 14); // lookupListOffset
02250     { // ScriptList table
02251         cacheU16 (gpos, 0); // scriptCount
02252     }
02253     { // Feature List table
02254         cacheU16 (gpos, 0); // featureCount
02255     }
02256     { // Lookup List Table
02257         cacheU16 (gpos, 0); // lookupCount
02258     }
02259 }
02260
02261 /**
02262  * @brief Fill a "GSUB" font table.
02263  */

```

```

02264     The "GSUB" table contains information for glyph substitution.
02265
02266     @param[in,out] font The Font struct to which to add the table.
02267 */
02268 void
02269 fillGsubTable (Font *font)
02270 {
02271     Buffer *gsub = newBuffer (38);
02272     addTable (font, "GSUB", gsub);
02273     cacheU16 (gsub, 1); // majorVersion
02274     cacheU16 (gsub, 0); // minorVersion
02275     cacheU16 (gsub, 10); // scriptListOffset
02276     cacheU16 (gsub, 34); // featureListOffset
02277     cacheU16 (gsub, 36); // lookupListOffset
02278     { // ScriptList table
02279         cacheU16 (gsub, 2); // scriptCount
02280         { // scriptRecords[0]
02281             cacheBytes (gsub, "DFLT", 4); // scriptTag
02282             cacheU16 (gsub, 14); // scriptOffset
02283         }
02284         { // scriptRecords[1]
02285             cacheBytes (gsub, "thai", 4); // scriptTag
02286             cacheU16 (gsub, 14); // scriptOffset
02287         }
02288         { // Script table
02289             cacheU16 (gsub, 4); // defaultLangSysOffset
02290             cacheU16 (gsub, 0); // langSysCount
02291             { // Default Language System table
02292                 cacheU16 (gsub, 0); // lookupOrderOffset
02293                 cacheU16 (gsub, 0); // requiredFeatureIndex
02294                 cacheU16 (gsub, 0); // featureIndexCount
02295             }
02296         }
02297     }
02298     { // Feature List table
02299         cacheU16 (gsub, 0); // featureCount
02300     }
02301     { // Lookup List Table
02302         cacheU16 (gsub, 0); // lookupCount
02303     }
02304 }
02305
02306 /**
02307     @brief Cache a string as a big-ending UTF-16 surrogate pair.
02308
02309     This function encodes a UTF-8 string as a big-endian UTF-16
02310     surrogate pair.
02311
02312     @param[in,out] buf Pointer to a Buffer struct to update.
02313     @param[in] str The character array to encode.
02314 */
02315 void
02316 cacheStringAsUTF16BE (Buffer *buf, const char *str)
02317 {
02318     for (const char *p = str; *p; p++)
02319     {
02320         byte c = *p;
02321         if (c < 0x80)
02322         {
02323             cacheU16 (buf, c);
02324             continue;
02325         }
02326         int length = 1;
02327         byte mask = 0x40;
02328         for (; c & mask; mask >>= 1)
02329             length++;
02330         if (length == 1 || length > 4)
02331             fail ("Ill-formed UTF-8 sequence.");
02332         uint_fast32_t codePoint = c & (mask - 1);
02333         for (int i = 1; i < length; i++)
02334         {
02335             c = *p++;
02336             if ((c & 0xc0) != 0x80) // NUL checked here
02337                 fail ("Ill-formed UTF-8 sequence.");
02338             codePoint = (codePoint << 6) | (c & 0x3f);
02339         }
02340         const int lowerBits = length==2 ? 7 : length==3 ? 11 : 16;
02341         if (codePoint >> lowerBits == 0)
02342             fail ("Ill-formed UTF-8 sequence."); // sequence should be shorter
02343         if (codePoint >= 0xd800 && codePoint <= 0xdfff)
02344             fail ("Ill-formed UTF-8 sequence.");
    
```



```

02345     if (codePoint > 0x10ffff)
02346         fail ("Ill-formed UTF-8 sequence.");
02347     if (codePoint > 0xffff)
02348     {
02349         cacheU16 (buf, 0xd800 | (codePoint - 0x10000) » 10);
02350         cacheU16 (buf, 0xdc00 | (codePoint & 0x3ff));
02351     }
02352     else
02353         cacheU16 (buf, codePoint);
02354 }
02355 }
02356
02357 /**
02358  @brief Fill a "name" font table.
02359
02360  The "name" table contains name information, for example for Name IDs.
02361
02362  @param[in,out] font The Font struct to which to add the table.
02363  @param[in] names List of NameStrings.
02364 */
02365 void
02366 fillNameTable (Font *font, NameStrings nameStrings)
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);
02398 }
02399
02400 /**
02401  @brief Print program version string on stdout.
02402
02403  Print program version if invoked with the "--version" option,
02404  and then exit successfully.
02405 */
02406 void
02407 printVersion (void) {
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     exit (EXIT_SUCCESS);
02416 }
02417
02418 /**
02419  @brief Print help message to stdout and then exit.
02420
02421  Print help message if invoked with the "--help" option,
02422  and then exit successfully.
02423 */
02424 void
02425

```



```

02426 printHelp (void) {
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");
02441     printf ("For more information, consult the hex2otf(1) man page.\n\n");
02442
02443     exit (EXIT_SUCCESS);
02444 }
02445
02446 /**
02447  @brief Data structure to hold options for OpenType font output.
02448
02449  This data structure holds the status of options that can be
02450  specified as command line arguments for creating the output
02451  OpenType font file.
02452 */
02453 typedef struct Options
02454 {
02455     bool truetype, blankOutline, bitmap, gpos, gsub;
02456     int cff; // 0 = no CFF outline; 1 = use 'CFF' table; 2 = use 'CFF2' table
02457     const char *hex, *pos, *out; // file names
02458     NameStrings nameStrings; // indexed directly by Name IDs
02459 } Options;
02460
02461 /**
02462  @brief Match a command line option with its key for enabling.
02463
02464  @param[in] operand A pointer to the specified operand.
02465  @param[in] key Pointer to the option structure.
02466  @param[in] delimiter The delimiter to end searching.
02467  @return Pointer to the first character of the desired option.
02468 */
02469 const char *
02470 matchToken (const char *operand, const char *key, char delimiter)
02471 {
02472     while (*key)
02473         if (*operand++ != *key++)
02474             return NULL;
02475     if (!*operand || *operand++ == delimiter)
02476         return operand;
02477     return NULL;
02478 }
02479
02480 /**
02481  @brief Parse command line options.
02482
02483  Option      Data Type      Description
02484  -----
02485  truetype    bool           Generate TrueType outlines
02486  blankOutline bool           Generate blank outlines
02487  bitmap      bool           Generate embedded bitmap
02488  gpos        bool           Generate a dummy GPOS table
02489  gsub        bool           Generate a dummy GSUB table
02490  cff         int            Generate CFF 1 or CFF 2 outlines
02491  hex         const char *    Name of Unifont .hex file
02492  pos         const char *    Name of Unifont combining data file
02493  out         const char *    Name of output font file
02494  nameStrings NameStrings     Array of TrueType font Name IDs
02495
02496  @param[in] argv Pointer to array of command line options.
02497  @return Data structure to hold requested command line options.
02498 */
02499 Options
02500 parseOptions (char *const argv[const])
02501 {
02502     Options opt = {0}; // all options default to 0, false and NULL
02503     const char *format = NULL;
02504     struct StringArg
02505     {
02506         const char *const key;

```

```

02507     const char **const value;
02508 } strArgs[] =
02509 {
02510     {"hex", &opt.hex},
02511     {"pos", &opt.pos},
02512     {"out", &opt.out},
02513     {"format", &format},
02514     {NULL, NULL} // sentinel
02515 };
02516 for (char *const *argp = argv + 1; *argp; argp++)
02517 {
02518     const char *const arg = *argp;
02519     struct StringArg *p;
02520     const char *value = NULL;
02521     if (strcmp (arg, "--help") == 0)
02522         printHelp ();
02523     if (strcmp (arg, "--version") == 0)
02524         printVersion ();
02525     for (p = strArgs; p->key; p++)
02526         if ((value = matchToken (arg, p->key, '=')))
02527             break;
02528     if (p->key)
02529     {
02530         if (!*value)
02531             fail ("Empty argument: '%s'", p->key);
02532         if (*p->value)
02533             fail ("Duplicate argument: '%s'", p->key);
02534         *p->value = value;
02535     }
02536     else // shall be a name string
02537     {
02538         char *endptr;
02539         unsigned long id = strtoul (arg, &endptr, 10);
02540         if (endptr == arg || id >= MAX_NAME_IDS || *endptr != '=')
02541             fail ("Invalid argument: '%s'", arg);
02542         endptr++; // skip '='
02543         if (opt.nameStrings[id])
02544             fail ("Duplicate name ID: %lu.", id);
02545         opt.nameStrings[id] = endptr;
02546     }
02547 }
02548 if (!opt.hex)
02549     fail ("Hex file is not specified.");
02550 if (opt.pos && opt.pos[0] == '\0')
02551     opt.pos = NULL; // Position file is optional. Empty path means none.
02552 if (!opt.out)
02553     fail ("Output file is not specified.");
02554 if (!format)
02555     fail ("Format is not specified.");
02556 for (const NamePair *p = defaultNames; p->str; p++)
02557     if (!opt.nameStrings[p->id])
02558         opt.nameStrings[p->id] = p->str;
02559 bool cff = false, cff2 = false;
02560 struct Symbol
02561 {
02562     const char *const key;
02563     bool *const found;
02564 } symbols[] =
02565 {
02566     {"cff", &cff},
02567     {"cff2", &cff2},
02568     {"truetype", &opt.truetype},
02569     {"blank", &opt.blankOutline},
02570     {"bitmap", &opt.bitmap},
02571     {"gpos", &opt.gpos},
02572     {"gsub", &opt.gsub},
02573     {NULL, NULL} // sentinel
02574 };
02575 while (*format)
02576 {
02577     const struct Symbol *p;
02578     const char *next = NULL;
02579     for (p = symbols; p->key; p++)
02580         if ((next = matchToken (format, p->key, ',')))
02581             break;
02582     if (!p->key)
02583         fail ("Invalid format.");
02584     *p->found = true;
02585     format = next;
02586 }
02587 if (cff + cff2 + opt.truetype + opt.blankOutline > 1)

```

```

02588     fail ("At most one outline format can be accepted.");
02589     if (!(cff || cff2 || opt.truetype || opt.bitmap))
02590         fail ("Invalid format.");
02591     opt.cff = cff + cff2 * 2;
02592     return opt;
02593 }
02594 /**
02595  @brief The main function.
02596  @param[in] argc The number of command-line arguments.
02597  @param[in] argv The array of command-line arguments.
02600  @return EXIT_FAILURE upon fatal error, EXIT_SUCCESS otherwise.
02601  */
02602 int
02603 main (int argc, char *argv[])
02604 {
02605     initBuffers (16);
02606     atexit (cleanBuffers);
02607     Options opt = parseOptions (argv);
02608     Font font;
02609     font.tables = newBuffer (sizeof (Table) * 16);
02610     font.glyphs = newBuffer (sizeof (Glyph) * MAX_GLYPHS);
02611     readGlyphs (&font, opt.hex);
02612     sortGlyphs (&font);
02613     enum LocaFormat loca = LOCA_OFFSET16;
02614     uint_fast16_t maxPoints = 0, maxContours = 0;
02615     pixels_t xMin = 0;
02616     if (opt.pos)
02617         positionGlyphs (&font, opt.pos, &xMin);
02618     if (opt.gpos)
02619         fillGposTable (&font);
02620     if (opt.gsub)
02621         fillGsubTable (&font);
02622     if (opt.cff)
02623         fillCFF (&font, opt.cff, opt.nameStrings);
02624     if (opt.truetype)
02625         fillTrueType (&font, &loca, &maxPoints, &maxContours);
02626     if (opt.blankOutline)
02627         fillBlankOutline (&font);
02628     if (opt.bitmap)
02629         fillBitmap (&font);
02630     fillHeadTable (&font, loca, xMin);
02631     fillHheaTable (&font, xMin);
02632     fillMaxpTable (&font, opt.cff, maxPoints, maxContours);
02633     fillOS2Table (&font);
02634     fillNameTable (&font, opt.nameStrings);
02635     fillHmtxTable (&font);
02636     fillCmapTable (&font);
02637     fillPostTable (&font);
02638     organizeTables (&font, opt.cff);
02639     writeFont (&font, opt.cff, opt.out);
02640     return EXIT_SUCCESS;
02641 }

```

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 "17.0.01"`
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

Copyright © 2022 何志翔 (He Zhixiang)

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 "17.0.01"
```

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.](#)

```
00001 /**
00002  @file hex2otf.h
00003
00004  @brief hex2otf.h - Header file for hex2otf.c
00005
00006  @copyright Copyright © 2022 何志翔 (He Zhixiang)
00007
00008  @author 何志翔 (He Zhixiang)
00009 */
00010
00011 /*
00012  LICENSE:
00013
00014  This program is free software; you can redistribute it and/or
00015  modify it under the terms of the GNU General Public License
00016  as published by the Free Software Foundation; either version 2
00017  of the License, or (at your option) any later version.
00018
00019  This program is distributed in the hope that it will be useful,
```

```

00020 but WITHOUT ANY WARRANTY; without even the implied warranty of
00021 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00022 GNU General Public License for more details.
00023
00024 You should have received a copy of the GNU General Public License
00025 along with this program; if not, write to the Free Software
00026 Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
00027 02110-1301, USA.
00028
00029 NOTE: It is a violation of the license terms of this software
00030 to delete license and copyright information below if creating
00031 a font derived from Unifont glyphs.
00032 */
00033 #ifndef _HEX2OTF_H_
00034 #define _HEX2OTF_H_
00035
00036 #define UNIFONT_VERSION "17.0.01" ///< Current Unifont version.
00037
00038 /**
00039  Define default strings for some TrueType font NameID strings.
00040
00041  NameID  Description
00042  -----
00043      0  Copyright Notice
00044      1  Font Family
00045      2  Font Subfamily
00046      5  Version of the Name Table
00047     11  URL of the Font Vendor
00048     13  License Description
00049     14  License Information URL
00050
00051  Default NameID 0 string (Copyright Notice)
00052 */
00053 #define DEFAULT_ID0 "Copyright © 1998-2022 Roman Czyborra, Paul Hardy, \
00054 Qianqian Fang, Andrew Miller, Johnnie Weaver, David Corbett, \
00055 Nils Moskopp, Rebecca Bettencourt, et al."
00056
00057 #define DEFAULT_ID1 "Unifont" ///< Default NameID 1 string (Font Family)
00058 #define DEFAULT_ID2 "Regular" ///< Default NameID 2 string (Font Subfamily)
00059
00060 ///< Default NameID 5 string (Version of the Name Table)
00061 #define DEFAULT_ID5 "Version " UNIFONT_VERSION
00062
00063 ///< Default NameID 11 string (Font Vendor URL)
00064 #define DEFAULT_ID11 "https://unifoundry.com/unifont/"
00065
00066 ///< Default NameID 13 string (License Description)
00067 #define DEFAULT_ID13 "Dual license: SIL Open Font License version 1.1, \
00068 and GNU GPL version 2 or later with the GNU Font Embedding Exception."
00069
00070 ///< Default NameID 14 string (License Information URLs)
00071 #define DEFAULT_ID14 "http://unifoundry.com/LICENSE.txt, \
00072 https://scripts.sil.org/OFL"
00073
00074 /**
00075  @brief Data structure for a font ID number and name character string.
00076 */
00077 typedef struct NamePair
00078 {
00079     int id;
00080     const char *str;
00081 } NamePair;
00082
00083 ///< Macro to initialize name identifier codes to default values defined above.
00084 #define NAMEPAIR(n) {(n), DEFAULT_ID##n}
00085
00086 /**
00087  @brief Allocate array of NameID codes with default values.
00088
00089  This array contains the default values for several TrueType NameID
00090  strings, as defined above in this file. Strings are assigned using
00091  the NAMEPAIR macro defined above.
00092 */
00093 const NamePair defaultNames[] =
00094 {
00095     NAMEPAIR(0), // Copyright notice; required (used in CFF)
00096     NAMEPAIR(1), // Font family; required (used in CFF)
00097     NAMEPAIR(2), // Font subfamily
00098     NAMEPAIR(5), // Version of the name table
00099     NAMEPAIR(11), // URL of font vendor
00100     NAMEPAIR(13), // License description

```



```
00101  NAMEPAIR (14), // License information URL
00102  {0, NULL}    // Sentinel
00103 };
00104
00105 #undef NAMEPAIR
00106
00107 #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](#) (void)
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

Copyright © 2023 Paul Hardy

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 (void);
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 (strcmp (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 (strcmp (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 (strcmp (argv [arg_count], "-h", 2) == 0 ||
00101                 strcmp (argv [arg_count], "--help", 6) == 0) {
00102             print_help ();
00103             exit (EXIT_SUCCESS);
00104         }
00105         arg_count++;
00106     }
00107 }
```

```

00109
00110  /*
00111   Initialize entire glyph array to zeroes in case the input
00112   file skips over some code points.
00113  */
00114  for (codept = 0; codept < MAX_GLYPHS; codept++) {
00115      for (i = 0; i < 32; i++) hangul_base[codept][i] = 0;
00116  }
00117
00118  /*
00119   Read the entire "hangul-base.hex" file into an array
00120   organized as hangul_base[code_point][glyph_byte].
00121   The Hangul glyphs are 16 columns wide, which is
00122   two bytes, by 16 rows, for a total of 2 * 16 = 32 bytes.
00123  */
00124  max_codept = hangul_read_base8 (infp, hangul_base);
00125  if (max_codept > 0xFF) {
00126      fprintf (stderr, "\nWARNING: Hangul glyph range exceeds PUA space.\n\n");
00127  }
00128
00129  /*
00130   For each glyph in the Unicode Hangul Syllables block,
00131   form a composite glyph of choseong + jungseong +
00132   optional jongseong and output it in Unifont .hex format.
00133  */
00134  for (codept = 0xAC00; codept < 0xAC00 + 19 * 21 * 28; codept++) {
00135      hangul_decompose (codept, &initial, &medial, &final);
00136
00137      hangul_syllable (initial, medial, final, hangul_base, syllable);
00138
00139      fprintf (outfp, "%04X:", codept);
00140
00141      for (i = 0; i < 32; i++) {
00142          fprintf (outfp, "%02X", syllable[i]);
00143      }
00144      fputc ('\n', outfp);
00145  }
00146
00147  exit (EXIT_SUCCESS);
00148 }

```

Here is the call graph for this function:

5.7.2.2 print_help()

```

void print_help (
    void )

```

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 demonstrtrtes\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.

```

00001 /**
00002  @file johab2syllables.c
00003
00004  @brief Create the Unicode Hangul Syllables block from component letters.
00005
00006  This program reads in a "hangul-base.hex" file containing Hangul
00007  letters in Johab 6/3/1 format and outputs a Unifont .hex format
00008  file covering the Unicode Hangul Syllables range of U+AC00..U+D7A3.
00009
00010  @author Paul Hardy
00011
00012  @copyright Copyright © 2023 Paul Hardy
00013 */
00014 /*
00015  LICENSE:
00016
00017  This program is free software: you can redistribute it and/or modify
00018  it under the terms of the GNU General Public License as published by
00019  the Free Software Foundation, either version 2 of the License, or
00020  (at your option) any later version.
00021
00022  This program is distributed in the hope that it will be useful,
00023  but WITHOUT ANY WARRANTY; without even the implied warranty of
00024  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00025  GNU General Public License for more details.
00026
00027  You should have received a copy of the GNU General Public License
00028  along with this program. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030
00031 #include <stdio.h>
00032 #include <stdlib.h>
00033 #include <string.h>
00034
00035 #include "hangul.h"
00036
00037
00038 /**
00039  @brief The main function.
00040 */
00041 int
00042 main (int argc, char *argv[]) {
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 (void);
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 (strcmp (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 (strcmp (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 (strcmp (argv [arg_count], "-h", 2) == 0 ||
00101          strcmp (argv [arg_count], "--help", 6) == 0) {
00102     print_help ();
00103     exit (EXIT_SUCCESS);
00104 }
00105
00106     arg_count++;
00107 }
00108
00109
00110 /*
00111     Initialize entire glyph array to zeroes in case the input
00112     file skips over some code points.
00113 */
00114 for (codept = 0; codept < MAX_GLYPHS; codept++) {
00115     for (i = 0; i < 32; i++) hangul_base[codept][i] = 0;
00116 }
00117
00118 /*
00119     Read the entire "hangul-base.hex" file into an array
00120     organized as hangul_base [code_point][glyph_byte].
00121     The Hangul glyphs are 16 columns wide, which is
00122     two bytes, by 16 rows, for a total of 2 * 16 = 32 bytes.
00123 */
00124 max_codept = hangul_read_base8 (infp, hangul_base);
00125 if (max_codept > 0x8FFF) {
00126     fprintf (stderr, "\nWARNING: Hangul glyph range exceeds PUA space.\n\n");
00127 }
00128
00129 /*
00130     For each glyph in the Unicode Hangul Syllables block,
00131     form a composite glyph of choseong + jungseong +
00132     optional jongseong and output it in Unifont .hex format.
00133 */
00134 for (codept = 0xAC00; codept < 0xAC00 + 19 * 21 * 28; codept++) {
00135     hangul_decompose (codept, &initial, &medial, &final);
00136
00137     hangul_syllable (initial, medial, final, hangul_base, syllable);
00138
00139     fprintf (outfp, "%04X:", codept);
00140
00141     for (i = 0; i < 32; i++) {
00142         fprintf (outfp, "%02X", syllable[i]);
00143     }
00144     fputc ('\n', outfp);
00145 }
00146
00147     exit (EXIT_SUCCESS);
00148 }
00149
00150
00151 /**
00152     @brief Print a help message.
00153 */
00154 void
00155 print_help (void) {
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 }
00179

```

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 (void)`
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 (
    void )
```

The main function.

Returns

Exit status is always 0 (successful termination).

Definition at line 46 of file `unibdf2hex.c`.

```
00047 {
00048     int i;
00049     int digitsout; /* how many hex digits we output in a bitmap */
00050     int thispoint;
00051     char inbuf[MAXBUF];
00052     int bbxx, bbxy, bbxxoff, bbxyoff;
00053
00054     int descent=4; /* font descent wrt baseline */
00055     int startrow; /* row to start glyph */
00056     unsigned rowout;
00057
00058     while (fgets (inbuf, MAXBUF - 1, stdin) != NULL) {
00059         if (strcmp (inbuf, "ENCODING ", 9) == 0) {
00060             sscanf (&inbuf[9], "%d", &thispoint); /* get code point */
00061             /*
00062              * If we want this code point, get the BBX (bounding box) and
00063              * BITMAP information.
00064              */
00065             if ((thispoint >= 0x2E80 && thispoint <= 0x2EFF) || // CJK Radicals Supplement
00066                 (thispoint >= 0x2F00 && thispoint <= 0x2FDF) || // Kangxi Radicals
00067                 (thispoint >= 0x2FF0 && thispoint <= 0x2FFF) || // Ideographic Description Characters
00068                 (thispoint >= 0x3001 && thispoint <= 0x303F) || // CJK Symbols and Punctuation (U+3000 is a space)
00069                 (thispoint >= 0x3100 && thispoint <= 0x312F) || // Bopomofo
00070                 (thispoint >= 0x31A0 && thispoint <= 0x31BF) || // Bopomofo extend
00071                 (thispoint >= 0x31C0 && thispoint <= 0x31EF) || // CJK Strokes
00072                 (thispoint >= 0x3400 && thispoint <= 0x4DBF) || // CJK Unified Ideographs Extension A
00073                 (thispoint >= 0x4E00 && thispoint <= 0x9FCF) || // CJK Unified Ideographs
00074                 (thispoint >= 0xF900 && thispoint <= 0xFAFF)) // CJK Compatibility Ideographs
00075             {
00076                 while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
00077                     strcmp (inbuf, "BBX ", 4) != 0); /* find bounding box */
00078
00079                 sscanf (&inbuf[4], "%d %d %d %d", &bbxx, &bbxy, &bbxxoff, &bbxyoff);
00080                 while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
00081                     strcmp (inbuf, "BITMAP", 6) != 0); /* find bitmap start */
00082                 fprintf (stdout, "%04X:", thispoint);
00083                 digitsout = 0;
00084                 /* Print initial blank rows */
00085                 startrow = descent + bbxyoff + bbxy;
00086
00087                 /* Force everything to 16 pixels wide */
00088                 for (i = 16; i > startrow; i--) {
00089                     fprintf (stdout, "0000");
00090                     digitsout += 4;
00091                 }
00092                 while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
00093                     strcmp (inbuf, "END", 3) != 0) { /* copy bitmap until END */
00094                     sscanf (inbuf, "%X", &rowout);
00095                     /* Now force glyph to a 16x16 grid even if they'd fit in 8x16 */
00096                     if (bbxx <= 8) rowout <<= 8; /* shift left for 16x16 glyph */
00097                     rowout >>= bbxxoff;
00098                     fprintf (stdout, "%04X", rowout);
00099                     digitsout += 4;
00100                 }
00101
00102                 /* Pad for 16x16 glyph */
00103                 while (digitsout < 64) {
00104                     fprintf (stdout, "0000");
00105                     digitsout += 4;
00106                 }
00107                 fprintf (stdout, "\n");
00108             }
00109         }
00110     }
00111     exit (0);
00112 }
```


5.10 unibdf2hex.c

Go to the documentation of this file.

```

00001 /**
00002  @file unibdf2hex.c
00003
00004  @brief unibdf2hex - Convert a BDF file into a unifont.hex file
00005
00006  @author Paul Hardy, January 2008
00007
00008  @copyright Copyright (C) 2008, 2013 Paul Hardy
00009
00010  Note: currently this has hard-coded code points for glyphs extracted
00011  from Wen Quan Yi to create the Unifont source file "wqy.hex".
00012 */
00013 /*
00014  LICENSE:
00015
00016  This program is free software: you can redistribute it and/or modify
00017  it under the terms of the GNU General Public License as published by
00018  the Free Software Foundation, either version 2 of the License, or
00019  (at your option) any later version.
00020
00021  This program is distributed in the hope that it will be useful,
00022  but WITHOUT ANY WARRANTY; without even the implied warranty of
00023  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00024  GNU General Public License for more details.
00025
00026  You should have received a copy of the GNU General Public License
00027  along with this program. If not, see <http://www.gnu.org/licenses/>.
00028 */
00029
00030 #include <stdio.h>
00031 #include <stdlib.h>
00032 #include <string.h>
00033
00034 #define UNISTART 0x3400 ///< First Unicode code point to examine
00035 #define UNISTOP 0x4DBF ///< Last Unicode code point to examine
00036
00037 #define MAXBUF 256 ///< Maximum allowable input file line length - 1
00038
00039
00040 /**
00041  @brief The main function.
00042
00043  @return Exit status is always 0 (successful termination).
00044 */
00045 int
00046 main (void)
00047 {
00048     int i;
00049     int digitsout; /* how many hex digits we output in a bitmap */
00050     int thispoint;
00051     char inbuf[MAXBUF];
00052     int bbxx, bbxy, bbxxoff, bbxyoff;
00053
00054     int descent=4; /* font descent wrt baseline */
00055     int startrow; /* row to start glyph */
00056     unsigned rowout;
00057
00058     while (fgets (inbuf, MAXBUF - 1, stdin) != NULL) {
00059         if (strcmp (inbuf, "ENCODING ", 9) == 0) {
00060             sscanf (&inbuf[9], "%d", &thispoint); /* get code point */
00061             /*
00062              If we want this code point, get the BBX (bounding box) and
00063              BITMAP information.
00064             */
00065             if ((thispoint >= 0x2E80 && thispoint <= 0x2EFF) || /* CJK Radicals Supplement
00066                  (thispoint >= 0x2F00 && thispoint <= 0x2FDF) || /* Kangxi Radicals
00067                  (thispoint >= 0x2FF0 && thispoint <= 0x2FFF) || /* Ideographic Description Characters
00068                  (thispoint >= 0x3001 && thispoint <= 0x303F) || /* CJK Symbols and Punctuation (U+3000 is a space)
00069                  (thispoint >= 0x3100 && thispoint <= 0x312F) || /* Bopomofo
00070                  (thispoint >= 0x31A0 && thispoint <= 0x31BF) || /* Bopomofo extend
00071                  (thispoint >= 0x31C0 && thispoint <= 0x31EF) || /* CJK Strokes
00072                  (thispoint >= 0x3400 && thispoint <= 0x4DBF) || /* CJK Unified Ideographs Extension A
00073                  (thispoint >= 0x4E00 && thispoint <= 0x9FCF) || /* CJK Unified Ideographs
00074                  (thispoint >= 0xF900 && thispoint <= 0FAFF)) /* CJK Compatibility Ideographs
00075             {
00076                 while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&

```

```

00077         strcmp (inbuf, "BBX ", 4) != 0); /* find bounding box */
00078
00079         sscanf (&inbuf[4], "%d %d %d %d", &bbxx, &bbxy, &bbxxoff, &bbxyoff);
00080         while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
00081             strcmp (inbuf, "BITMAP", 6) != 0); /* find bitmap start */
00082         fprintf (stdout, "%04X:", thispoint);
00083         digitsout = 0;
00084         /* Print initial blank rows */
00085         startrow = descent + bbxyoff + bbxy;
00086
00087         /* Force everything to 16 pixels wide */
00088         for (i = 16; i > startrow; i--) {
00089             fprintf (stdout, "0000");
00090             digitsout += 4;
00091         }
00092         while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
00093             strcmp (inbuf, "END", 3) != 0) { /* copy bitmap until END */
00094             sscanf (inbuf, "%X", &rowout);
00095             /* Now force glyph to a 16x16 grid even if they'd fit in 8x16 */
00096             if (bbxx <= 8) rowout <= 8; /* shift left for 16x16 glyph */
00097             rowout >= bbxxoff;
00098             fprintf (stdout, "%04X", rowout);
00099             digitsout += 4;
00100         }
00101
00102         /* Pad for 16x16 glyph */
00103         while (digitsout < 64) {
00104             fprintf (stdout, "0000");
00105             digitsout += 4;
00106         }
00107         fprintf (stdout, "\n");
00108     }
00109 }
00110 }
00111 exit (0);
00112 }

```

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.
- 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

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

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

Definition in file `unibmp2hex.c`.

5.11.2 Macro Definition Documentation

5.11.2.1 MAXBUF

```
#define MAXBUF 256
```

Maximum input file line length - 1.

Definition at line [121](#) 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 [166](#) of file [unibmp2hex.c](#).

```
00167 {
00168
00169     int i, j, k; /* loop variables */
00170     unsigned char inchar; /* temporary input character */
00171     char header[MAXBUF]; /* input buffer for bitmap file header */
00172     int wbmp=0; /* =0 for Windows Bitmap (.bmp); 1 for Wireless Bitmap (.wbmp) */
00173     int fatal; /* =1 if a fatal error occurred */
00174     int match; /* =1 if we're still matching a pattern, 0 if no match */
00175     int empty1, empty2; /* =1 if bytes tested are all zeroes */
00176     unsigned char thischar1[16], thischar2[16]; /* bytes of hex char */
00177     unsigned char thischar0[16], thischar3[16]; /* bytes for quadruple-width */
00178     int thisrow; /* index to point into thischar1[] and thischar2[] */
00179     int tmpsum; /* temporary sum to see if a character is blank */
00180     unsigned this_pixel; /* color of one pixel, if > 1 bit per pixel */
00181     unsigned next_pixels; /* pending group of 8 pixels being read */
00182     unsigned color_mask = 0x00; /* to invert monochrome bitmap, set to 0xFF */
00183 }
```

```

00184 unsigned char bitmap[17*32][18*32/8]; /* final bitmap */
00185 /* For wide array:
00186     0 = don't force glyph to double-width;
00187     1 = force glyph to double-width;
00188     4 = force glyph to quadruple-width.
00189 */
00190 char wide[0x200000]={0x200000 * 0};
00191
00192 char *infile="", *outfile=""; /* names of input and output files */
00193 FILE *infp, *outfp; /* file pointers of input and output files */
00194
00195 if (argc > 1) {
00196     for (i = 1; i < argc; i++) {
00197         if (argv[i][0] == '-') { /* this is an option argument */
00198             switch (argv[i][1]) {
00199                 case 'i': /* name of input file */
00200                     infile = &argv[i][2];
00201                     break;
00202                 case 'o': /* name of output file */
00203                     outfile = &argv[i][2];
00204                     break;
00205                 case 'p': /* specify a Unicode plane */
00206                     sscanf (&argv[i][2], "%x", &uniplane); /* Get Unicode plane */
00207                     planeset = 1; /* Use specified range, not what's in bitmap */
00208                     break;
00209                 case 'w': /* force wide (16 pixels) for each glyph */
00210                     forcewide = 1;
00211                     break;
00212                 default: /* if unrecognized option, print list and exit */
00213                     fprintf (stderr, "\nSyntax:\n\n");
00214                     fprintf (stderr, " %s -p<Unicode_Page> ", argv[0]);
00215                     fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
00216                     fprintf (stderr, " -w specifies .wbmp output instead of ");
00217                     fprintf (stderr, "default Windows .bmp output.\n\n");
00218                     fprintf (stderr, " -p is followed by 1 to 6 ");
00219                     fprintf (stderr, "Unicode plane hex digits ");
00220                     fprintf (stderr, "(default is Page 0).\n\n");
00221                     fprintf (stderr, "\nExample:\n\n");
00222                     fprintf (stderr, " %s -p83 -iunifont.hex -ou83.bmp\n\n",
00223                             argv[0]);
00224                     exit (1);
00225             }
00226         }
00227     }
00228 }
00229 /*
00230     Make sure we can open any I/O files that were specified before
00231     doing anything else.
00232 */
00233 if (strlen (infile) > 0) {
00234     if ((infp = fopen (infile, "r")) == NULL) {
00235         fprintf (stderr, "Error: can't open %s for input.\n", infile);
00236         exit (1);
00237     }
00238 }
00239 else {
00240     infp = stdin;
00241 }
00242 if (strlen (outfile) > 0) {
00243     if ((outfp = fopen (outfile, "w")) == NULL) {
00244         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00245         exit (1);
00246     }
00247 }
00248 else {
00249     outfp = stdout;
00250 }
00251 /*
00252     Initialize selected code points for double width (16x16).
00253     Double-width is forced in cases where a glyph (usually a combining
00254     glyph) only occupies the left-hand side of a 16x16 grid, but must
00255     be rendered as double-width to appear properly with other glyphs
00256     in a given script. If additions were made to a script after
00257     Unicode 5.0, the Unicode version is given in parentheses after
00258     the script name.
00259 */
00260 for (i = 0x0700; i <= 0x074F; i++) wide[i] = 1; /* Syriac */
00261 for (i = 0x0800; i <= 0x083F; i++) wide[i] = 1; /* Samaritan (5.2) */
00262 for (i = 0x0900; i <= 0x0DFF; i++) wide[i] = 1; /* Indic */
00263 for (i = 0x1000; i <= 0x109F; i++) wide[i] = 1; /* Myanmar */
00264 for (i = 0x1100; i <= 0x11FF; i++) wide[i] = 1; /* Hangul Jamo */

```

```

00265 for (i = 0x1400; i <= 0x167F; i++) wide[i] = 1; /* Canadian Aboriginal */
00266 for (i = 0x1700; i <= 0x171F; i++) wide[i] = 1; /* Tagalog */
00267 for (i = 0x1720; i <= 0x173F; i++) wide[i] = 1; /* Hanunoo */
00268 for (i = 0x1740; i <= 0x175F; i++) wide[i] = 1; /* Buhid */
00269 for (i = 0x1760; i <= 0x177F; i++) wide[i] = 1; /* Tagbanwa */
00270 for (i = 0x1780; i <= 0x17FF; i++) wide[i] = 1; /* Khmer */
00271 for (i = 0x18B0; i <= 0x18FF; i++) wide[i] = 1; /* Ext. Can. Aboriginal */
00272 for (i = 0x1800; i <= 0x18AF; i++) wide[i] = 1; /* Mongolian */
00273 for (i = 0x1900; i <= 0x194F; i++) wide[i] = 1; /* Limbu */
00274 // for (i = 0x1980; i <= 0x19DF; i++) wide[i] = 1; /* New Tai Lue */
00275 for (i = 0x1A00; i <= 0x1A1F; i++) wide[i] = 1; /* Buginese */
00276 for (i = 0x1A20; i <= 0x1AAF; i++) wide[i] = 1; /* Tai Tham (5.2) */
00277 for (i = 0x1B00; i <= 0x1B7F; i++) wide[i] = 1; /* Balinese */
00278 for (i = 0x1B80; i <= 0x1BBF; i++) wide[i] = 1; /* Sundanese (5.1) */
00279 for (i = 0x1BC0; i <= 0x1BFF; i++) wide[i] = 1; /* Batak (6.0) */
00280 for (i = 0x1C00; i <= 0x1C4F; i++) wide[i] = 1; /* Lepcha (5.1) */
00281 for (i = 0x1CC0; i <= 0x1CCF; i++) wide[i] = 1; /* Sundanese Supplement */
00282 for (i = 0x1CD0; i <= 0x1CFF; i++) wide[i] = 1; /* Vedic Extensions (5.2) */
00283 wide[0x2329] = wide[0x232A] = 1; /* Left- & Right-pointing Angle Brackets */
00284 for (i = 0x2E80; i <= 0xA4CF; i++) wide[i] = 1; /* CJK */
00285 // for (i = 0x9FD8; i <= 0x9FE9; i++) wide[i] = 4; /* CJK quadruple-width */
00286 for (i = 0xA900; i <= 0xA92F; i++) wide[i] = 1; /* Kayah Li (5.1) */
00287 for (i = 0xA930; i <= 0xA95F; i++) wide[i] = 1; /* Rejang (5.1) */
00288 for (i = 0xA960; i <= 0xA97F; i++) wide[i] = 1; /* Hangul Jamo Extended-A */
00289 for (i = 0xA980; i <= 0xA9DF; i++) wide[i] = 1; /* Javanese (5.2) */
00290 for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham (5.1) */
00291 for (i = 0xA9E0; i <= 0xA9FF; i++) wide[i] = 1; /* Myanmar Extended-B */
00292 for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham */
00293 for (i = 0xAA60; i <= 0xAA7F; i++) wide[i] = 1; /* Myanmar Extended-A */
00294 for (i = 0xAAE0; i <= 0xA AFF; i++) wide[i] = 1; /* Meetei Mayek Ext (6.0) */
00295 for (i = 0xABC0; i <= 0xABFF; i++) wide[i] = 1; /* Meetei Mayek (5.2) */
00296 for (i = 0xAC00; i <= 0xD7AF; i++) wide[i] = 1; /* Hangul Syllables */
00297 for (i = 0xD7B0; i <= 0xD7FF; i++) wide[i] = 1; /* Hangul Jamo Extended-B */
00298 for (i = 0xF900; i <= 0FAFF; i++) wide[i] = 1; /* CJK Compatibility */
00299 for (i = 0xFE10; i <= 0xFE1F; i++) wide[i] = 1; /* Vertical Forms */
00300 for (i = 0xFE30; i <= 0xFE60; i++) wide[i] = 1; /* CJK Compatibility Forms */
00301 for (i = 0xFFE0; i <= 0xFFE6; i++) wide[i] = 1; /* CJK Compatibility Forms */
00302
00303 wide[0x303F] = 0; /* CJK half-space fill */
00304
00305 /* Supplemental Multilingual Plane (Plane 01) */
00306 for (i = 0x0105C0; i <= 0x0105FF; i++) wide[i] = 1; /* Toghri */
00307 for (i = 0x010A00; i <= 0x010A5F; i++) wide[i] = 1; /* Kharoshthi */
00308 for (i = 0x011000; i <= 0x01107F; i++) wide[i] = 1; /* Brahmi */
00309 for (i = 0x011080; i <= 0x0110CF; i++) wide[i] = 1; /* Kaithi */
00310 for (i = 0x011100; i <= 0x01114F; i++) wide[i] = 1; /* Chakma */
00311 for (i = 0x011180; i <= 0x0111DF; i++) wide[i] = 1; /* Sharada */
00312 for (i = 0x011200; i <= 0x01124F; i++) wide[i] = 1; /* Khojki */
00313 for (i = 0x0112B0; i <= 0x0112FF; i++) wide[i] = 1; /* Khudawadi */
00314 for (i = 0x011300; i <= 0x01137F; i++) wide[i] = 1; /* Grantha */
00315 for (i = 0x011380; i <= 0x0113FF; i++) wide[i] = 1; /* Tulu-Tigalari */
00316 for (i = 0x011400; i <= 0x01147F; i++) wide[i] = 1; /* Newa */
00317 for (i = 0x011480; i <= 0x0114DF; i++) wide[i] = 1; /* Tirhuta */
00318 for (i = 0x011580; i <= 0x0115FF; i++) wide[i] = 1; /* Siddham */
00319 for (i = 0x011600; i <= 0x01165F; i++) wide[i] = 1; /* Modi */
00320 for (i = 0x011660; i <= 0x01167F; i++) wide[i] = 1; /* Mongolian Suppl. */
00321 for (i = 0x011680; i <= 0x0116CF; i++) wide[i] = 1; /* Takri */
00322 for (i = 0x0116D0; i <= 0x0116FF; i++) wide[i] = 1; /* Myanmar Extended-C */
00323 for (i = 0x011700; i <= 0x01173F; i++) wide[i] = 1; /* Ahom */
00324 for (i = 0x011800; i <= 0x01184F; i++) wide[i] = 1; /* Dogra */
00325 for (i = 0x011900; i <= 0x01195F; i++) wide[i] = 1; /* Dives Akuru */
00326 for (i = 0x0119A0; i <= 0x0119FF; i++) wide[i] = 1; /* Nandinagari */
00327 for (i = 0x011A00; i <= 0x011A4F; i++) wide[i] = 1; /* Zanabazar Square */
00328 for (i = 0x011A50; i <= 0x011AAF; i++) wide[i] = 1; /* Soyombo */
00329 for (i = 0x011B00; i <= 0x011B5F; i++) wide[i] = 1; /* Devanagari Extended-A */
00330 for (i = 0x011B60; i <= 0x011B7F; i++) wide[i] = 1; /* Sharada Supplement */
00331 for (i = 0x011F00; i <= 0x011F5F; i++) wide[i] = 1; /* Kawi */
00332 for (i = 0x011C00; i <= 0x011C6F; i++) wide[i] = 1; /* Bhaiksuki */
00333 for (i = 0x011C70; i <= 0x011CBF; i++) wide[i] = 1; /* Marchen */
00334 for (i = 0x011D00; i <= 0x011D5F; i++) wide[i] = 1; /* Masaram Gondi */
00335 for (i = 0x011EE0; i <= 0x011EFF; i++) wide[i] = 1; /* Makasar */
00336 for (i = 0x011F00; i <= 0x011F5F; i++) wide[i] = 1; /* Kawi */
00337 for (i = 0x012F90; i <= 0x012FFF; i++) wide[i] = 1; /* Cypro-Minoan */
00338 /* Make Bassa Vah all single width or all double width */
00339 for (i = 0x016100; i <= 0x01613F; i++) wide[i] = 1; /* Gurung Khema */
00340 for (i = 0x016AD0; i <= 0x016AFF; i++) wide[i] = 1; /* Bassa Vah */
00341 for (i = 0x016B00; i <= 0x016B8F; i++) wide[i] = 1; /* Pahawh Hmong */
00342 for (i = 0x016D40; i <= 0x016D7F; i++) wide[i] = 1; /* Kirat Rai */
00343 for (i = 0x016F00; i <= 0x016F9F; i++) wide[i] = 1; /* Miao */
00344 for (i = 0x016FE0; i <= 0x016FFF; i++) wide[i] = 1; /* Ideograph Sym/Punct */
00345 for (i = 0x017000; i <= 0x0187FF; i++) wide[i] = 1; /* Tangut */

```

```

00346 for (i = 0x018800; i <= 0x018AFF; i++) wide[i] = 1; /* Tangut Components */
00347 for (i = 0x018B00; i <= 0x018CFF; i++) wide[i] = 1; /* Khitan Small Script */
00348 for (i = 0x01AFF0; i <= 0x01AFFF; i++) wide[i] = 1; /* Kana Extended-B */
00349 for (i = 0x01B000; i <= 0x01B0FF; i++) wide[i] = 1; /* Kana Supplement */
00350 for (i = 0x01B100; i <= 0x01B12F; i++) wide[i] = 1; /* Kana Extended-A */
00351 for (i = 0x01B170; i <= 0x01B2FF; i++) wide[i] = 1; /* Nushu */
00352 for (i = 0x01CF00; i <= 0x01CFCF; i++) wide[i] = 1; /* Znamenny Musical */
00353 for (i = 0x01D800; i <= 0x01DAAF; i++) wide[i] = 1; /* Sutton SignWriting */
00354 for (i = 0x01E500; i <= 0x01E5FF; i++) wide[i] = 1; /* Ol Onal */
00355 for (i = 0x01E800; i <= 0x01E8DF; i++) wide[i] = 1; /* Mende Kikakui */
00356 for (i = 0x01F200; i <= 0x01F2FF; i++) wide[i] = 1; /* Encl Ideograp Suppl */
00357 wide[0x01F5E7] = 1; /* Three Rays Right */
00358
00359 /*
00360 Determine whether or not the file is a Microsoft Windows Bitmap file.
00361 If it starts with 'B', 'M', assume it's a Windows Bitmap file.
00362 Otherwise, assume it's a Wireless Bitmap file.
00363
00364 WARNING: There isn't much in the way of error checking here --
00365 if you give it a file that wasn't first created by hex2bmp.c,
00366 all bets are off.
00367 */
00368 fatal = 0; /* assume everything is okay with reading input file */
00369 if ((header[0] = fgetc (infp)) != EOF) {
00370     if ((header[1] = fgetc (infp)) != EOF) {
00371         if (header[0] == 'B' && header[1] == 'M') {
00372             wbmp = 0; /* Not a Wireless Bitmap -- it's a Windows Bitmap */
00373         }
00374         else {
00375             wbmp = 1; /* Assume it's a Wireless Bitmap */
00376         }
00377     }
00378     else
00379         fatal = 1;
00380 }
00381 else
00382     fatal = 1;
00383
00384 if (fatal) {
00385     fprintf (stderr, "Fatal error; end of input file.\n\n");
00386     exit (1);
00387 }
00388 /*
00389 If this is a Wireless Bitmap (.wbmp) format file,
00390 skip the header and point to the start of the bitmap itself.
00391 */
00392 if (wbmp) {
00393     for (i=2; i<6; i++)
00394         header[i] = fgetc (infp);
00395     /*
00396     Now read the bitmap.
00397     */
00398     for (i=0; i < 32*17; i++) {
00399         for (j=0; j < 32*18/8; j++) {
00400             inchar = fgetc (infp);
00401             bitmap[i][j] = ~inchar; /* invert bits for proper color */
00402         }
00403     }
00404 }
00405 /*
00406 Otherwise, treat this as a Windows Bitmap file, because we checked
00407 that it began with "BM". Save the header contents for future use.
00408 Expect a 14 byte standard BITMAPFILEHEADER format header followed
00409 by a 40 byte standard BITMAPINFOHEADER Device Independent Bitmap
00410 header, with data stored in little-endian format.
00411 */
00412 else {
00413     for (i = 2; i < 54; i++)
00414         header[i] = fgetc (infp);
00415
00416     bmp_header.filetype[0] = 'B';
00417     bmp_header.filetype[1] = 'M';
00418
00419     bmp_header.file_size =
00420         (header[2] & 0xFF) | ((header[3] & 0xFF) << 8) |
00421         ((header[4] & 0xFF) << 16) | ((header[5] & 0xFF) << 24);
00422
00423     /* header bytes 6..9 are reserved */
00424
00425     bmp_header.image_offset =
00426         (header[10] & 0xFF) | ((header[11] & 0xFF) << 8) |

```

```

00427     ((header[12] & 0xFF) « 16) | ((header[13] & 0xFF) « 24);
00428
00429     bmp_header.info_size =
00430         (header[14] & 0xFF) | ((header[15] & 0xFF) « 8) |
00431         ((header[16] & 0xFF) « 16) | ((header[17] & 0xFF) « 24);
00432
00433     bmp_header.width =
00434         (header[18] & 0xFF) | ((header[19] & 0xFF) « 8) |
00435         ((header[20] & 0xFF) « 16) | ((header[21] & 0xFF) « 24);
00436
00437     bmp_header.height =
00438         (header[22] & 0xFF) | ((header[23] & 0xFF) « 8) |
00439         ((header[24] & 0xFF) « 16) | ((header[25] & 0xFF) « 24);
00440
00441     bmp_header.nplanes =
00442         (header[26] & 0xFF) | ((header[27] & 0xFF) « 8);
00443
00444     bmp_header.bits_per_pixel =
00445         (header[28] & 0xFF) | ((header[29] & 0xFF) « 8);
00446
00447     bmp_header.compression =
00448         (header[30] & 0xFF) | ((header[31] & 0xFF) « 8) |
00449         ((header[32] & 0xFF) « 16) | ((header[33] & 0xFF) « 24);
00450
00451     bmp_header.image_size =
00452         (header[34] & 0xFF) | ((header[35] & 0xFF) « 8) |
00453         ((header[36] & 0xFF) « 16) | ((header[37] & 0xFF) « 24);
00454
00455     bmp_header.x_ppm =
00456         (header[38] & 0xFF) | ((header[39] & 0xFF) « 8) |
00457         ((header[40] & 0xFF) « 16) | ((header[41] & 0xFF) « 24);
00458
00459     bmp_header.y_ppm =
00460         (header[42] & 0xFF) | ((header[43] & 0xFF) « 8) |
00461         ((header[44] & 0xFF) « 16) | ((header[45] & 0xFF) « 24);
00462
00463     bmp_header.ncolors =
00464         (header[46] & 0xFF) | ((header[47] & 0xFF) « 8) |
00465         ((header[48] & 0xFF) « 16) | ((header[49] & 0xFF) « 24);
00466
00467     bmp_header.important_colors =
00468         (header[50] & 0xFF) | ((header[51] & 0xFF) « 8) |
00469         ((header[52] & 0xFF) « 16) | ((header[53] & 0xFF) « 24);
00470
00471     if (bmp_header.ncolors == 0)
00472         bmp_header.ncolors = 1 « bmp_header.bits_per_pixel;
00473
00474     /* If a Color Table exists, read it */
00475     if (bmp_header.ncolors > 0 && bmp_header.bits_per_pixel <= 8) {
00476         for (i = 0; i < bmp_header.ncolors; i++) {
00477             color_table[i][0] = fgetc (infp); /* Red */
00478             color_table[i][1] = fgetc (infp); /* Green */
00479             color_table[i][2] = fgetc (infp); /* Blue */
00480             color_table[i][3] = fgetc (infp); /* Alpha */
00481         }
00482         /*
00483          Determine from the first color table entry whether we
00484          are inverting the resulting bitmap image.
00485          */
00486         if ( (color_table[0][0] + color_table[0][1] + color_table[0][2])
00487              < (3 * 128) ) {
00488             color_mask = 0xFF;
00489         }
00490     }
00491
00492 #ifdef DEBUG
00493
00494     /*
00495      Print header info for possibly adding support for
00496      additional file formats in the future, to determine
00497      how the bitmap is encoded.
00498      */
00499     fprintf (stderr, "Filetype: '%c%c'\n",
00500             bmp_header.filetype[0], bmp_header.filetype[1]);
00501     fprintf (stderr, "File Size: %d\n", bmp_header.file_size);
00502     fprintf (stderr, "Image Offset: %d\n", bmp_header.image_offset);
00503     fprintf (stderr, "Info Header Size: %d\n", bmp_header.info_size);
00504     fprintf (stderr, "Image Width: %d\n", bmp_header.width);
00505     fprintf (stderr, "Image Height: %d\n", bmp_header.height);
00506     fprintf (stderr, "Number of Planes: %d\n", bmp_header.nplanes);
00507     fprintf (stderr, "Bits per Pixel: %d\n", bmp_header.bits_per_pixel);

```



```

00508     fprintf(stderr, "Compression Method: %d\n", bmp_header.compression);
00509     fprintf(stderr, "Image Size: %d\n", bmp_header.image_size);
00510     fprintf(stderr, "X Pixels per Meter: %d\n", bmp_header.x_ppm);
00511     fprintf(stderr, "Y Pixels per Meter: %d\n", bmp_header.y_ppm);
00512     fprintf(stderr, "Number of Colors: %d\n", bmp_header.ncolors);
00513     fprintf(stderr, "Important Colors: %d\n", bmp_header.important_colors);
00514
00515 #endif
00516
00517     /*
00518     Now read the bitmap.
00519     */
00520     for (i = 32*17-1; i >= 0; i--) {
00521         for (j=0; j < 32*18/8; j++) {
00522             next_pixels = 0x00; /* initialize next group of 8 pixels */
00523             /* Read a monochrome image -- the original case */
00524             if (bmp_header.bits_per_pixel == 1) {
00525                 next_pixels = fgetc (infp);
00526             }
00527             /* Read a 32 bit per pixel RGB image; convert to monochrome */
00528             else if ( bmp_header.bits_per_pixel == 24 ||
00529                     bmp_header.bits_per_pixel == 32) {
00530                 next_pixels = 0;
00531                 for (k = 0; k < 8; k++) { /* get next 8 pixels */
00532                     this_pixel = (fgetc (infp) & 0xFF) +
00533                                 (fgetc (infp) & 0xFF) +
00534                                 (fgetc (infp) & 0xFF);
00535
00536                     if (bmp_header.bits_per_pixel == 32) {
00537                         (void) fgetc (infp); /* ignore alpha value */
00538                     }
00539
00540                     /* convert RGB color space to monochrome */
00541                     if (this_pixel >= (128 * 3))
00542                         this_pixel = 0;
00543                     else
00544                         this_pixel = 1;
00545
00546                     /* shift next pixel color into place for 8 pixels total */
00547                     next_pixels = (next_pixels « 1) | this_pixel;
00548                 }
00549             }
00550             if (bmp_header.height < 0) { /* Bitmap drawn top to bottom */
00551                 bitmap [(32*17-1) - i] [j] = next_pixels;
00552             }
00553             else { /* Bitmap drawn bottom to top */
00554                 bitmap [i] [j] = next_pixels;
00555             }
00556         }
00557     }
00558
00559     /*
00560     If any bits are set in color_mask, apply it to
00561     entire bitmap to invert black <--> white.
00562     */
00563     if (color_mask != 0x00) {
00564         for (i = 32*17-1; i >= 0; i--) {
00565             for (j=0; j < 32*18/8; j++) {
00566                 bitmap [i] [j] ^= color_mask;
00567             }
00568         }
00569     }
00570
00571 }
00572
00573 /*
00574 We've read the entire file. Now close the input file pointer.
00575 */
00576 fclose (infp);
00577 /*
00578 We now have the header portion in the header[] array,
00579 and have the bitmap portion from top-to-bottom in the bitmap[] array.
00580 */
00581 /*
00582 If no Unicode range (U+nnnnnn00 through U+nnnnnnFF) was specified
00583 with a -p parameter, determine the range from the digits in the
00584 bitmap itself.
00585
00586 Store bitmaps for the hex digit patterns that this file uses.
00587 */
00588 if (!planeset) { /* If Unicode range not specified with -p parameter */

```

```

00589     for (i = 0x0; i <= 0xF; i++) { /* hex digit pattern we're storing */
00590         for (j = 0; j < 4; j++) {
00591             hexdigit[i][j] =
00592                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8][6] << 24) |
00593                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 1][6] << 16) |
00594                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 2][6] << 8) |
00595                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 3][6]);
00596         }
00597     }
00598     /*
00599     Read the Unicode plane digits into arrays for comparison, to
00600     determine the upper four hex digits of the glyph addresses.
00601     */
00602     for (i = 0; i < 4; i++) {
00603         for (j = 0; j < 4; j++) {
00604             unidigit[i][j] =
00605                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 1][i + 3] << 24) |
00606                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 2][i + 3] << 16) |
00607                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 3][i + 3] << 8) |
00608                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 4][i + 3]);
00609         }
00610     }
00611
00612     tmpsum = 0;
00613     for (i = 4; i < 6; i++) {
00614         for (j = 0; j < 4; j++) {
00615             unidigit[i][j] =
00616                 ((unsigned)bitmap[32 * 1 + 4 * j + 8][i] << 24) |
00617                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 1][i] << 16) |
00618                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 2][i] << 8) |
00619                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 3][i]);
00620             tmpsum |= unidigit[i][j];
00621         }
00622     }
00623     if (tmpsum == 0) { /* the glyph matrix is transposed */
00624         flip = 1; /* note transposed order for processing glyphs in matrix */
00625         /*
00626         Get 5th and 6th hex digits by shifting first column header left by
00627         1.5 columns, thereby shifting the hex digit right after the leading
00628         "U+nnnn" page number.
00629         */
00630         for (i = 0x08; i < 0x18; i++) {
00631             bitmap[i][7] = (bitmap[i][8] << 4) | ((bitmap[i][9] >> 4) & 0xf);
00632             bitmap[i][8] = (bitmap[i][9] << 4) | ((bitmap[i][10] >> 4) & 0xf);
00633         }
00634         for (i = 4; i < 6; i++) {
00635             for (j = 0; j < 4; j++) {
00636                 unidigit[i][j] =
00637                     ((unsigned)bitmap[4 * j + 8 + 1][i + 3] << 24) |
00638                     ((unsigned)bitmap[4 * j + 8 + 2][i + 3] << 16) |
00639                     ((unsigned)bitmap[4 * j + 8 + 3][i + 3] << 8) |
00640                     ((unsigned)bitmap[4 * j + 8 + 4][i + 3]);
00641             }
00642         }
00643     }
00644
00645     /*
00646     Now determine the Unicode plane by comparing unidigit[0..5] to
00647     the hexdigit[0x0..0xF] array.
00648     */
00649     uniplane = 0;
00650     for (i=0; i<6; i++) { /* go through one bitmap digit at a time */
00651         match = 0; /* haven't found pattern yet */
00652         for (j = 0x0; !match && j <= 0xF; j++) {
00653             if (unidigit[i][0] == hexdigit[j][0] &&
00654                 unidigit[i][1] == hexdigit[j][1] &&
00655                 unidigit[i][2] == hexdigit[j][2] &&
00656                 unidigit[i][3] == hexdigit[j][3]) { /* we found the digit */
00657                 uniplane |= j;
00658                 match = 1;
00659             }
00660         }
00661         uniplane <<= 4;
00662     }
00663     uniplane >>= 4;
00664 }
00665 /*
00666 Now read each glyph and print it as hex.
00667 */
00668 for (i = 0x0; i <= 0xf; i++) {
00669     for (j = 0x0; j <= 0xf; j++) {

```

```

00670     for (k = 0; k < 16; k++) {
00671         if (flip) { /* transpose glyph matrix */
00672             thischar0[k] = bitmap[32*(j+1) + k + 7][4 * (i+2)   ];
00673             thischar1[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 1];
00674             thischar2[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 2];
00675             thischar3[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 3];
00676         }
00677         else {
00678             thischar0[k] = bitmap[32*(i+1) + k + 7][4 * (j+2)   ];
00679             thischar1[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 1];
00680             thischar2[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 2];
00681             thischar3[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 3];
00682         }
00683     }
00684     /*
00685     If the second half of the 16*16 character is all zeroes, this
00686     character is only 8 bits wide, so print a half-width character.
00687     */
00688     empty1 = empty2 = 1;
00689     for (k=0; (empty1 || empty2) && k < 16; k++) {
00690         if (thischar1[k] != 0) empty1 = 0;
00691         if (thischar2[k] != 0) empty2 = 0;
00692     }
00693     /*
00694     Only print this glyph if it isn't blank.
00695     */
00696     if (!empty1 || !empty2) {
00697         /*
00698         If the second half is empty, this is a half-width character.
00699         Only print the first half.
00700         */
00701         /*
00702         Original GNU Unifont format is four hexadecimal digit character
00703         code followed by a colon followed by a hex string. Add support
00704         for codes beyond the Basic Multilingual Plane.
00705
00706         Unicode ranges from U+0000 to U+10FFFF, so print either a
00707         4-digit or a 6-digit code point. Note that this software
00708         should support up to an 8-digit code point, extending beyond
00709         the normal Unicode range, but this has not been fully tested.
00710         */
00711         if (uniplane > 0xff)
00712             fprintf (outfp, "%04X%X%X:X:", uniplane, i, j); // 6 digit code pt.
00713         else
00714             fprintf (outfp, "%02X%X%X:X:", uniplane, i, j); // 4 digit code pt.
00715         for (thisrow=0; thisrow<16; thisrow++) {
00716             /*
00717             If second half is empty and we're not forcing this
00718             code point to double width, print as single width.
00719             */
00720             if (!forcewide &&
00721                 empty2 && !wide[(uniplane « 8) | (i « 4) | j]) {
00722                 fprintf (outfp,
00723                     "%02X",
00724                     thischar1[thisrow]);
00725             }
00726             else if (wide[(uniplane « 8) | (i « 4) | j] == 4) {
00727                 /* quadruple-width; force 32nd pixel to zero */
00728                 fprintf (outfp,
00729                     "%02X%02X%02X%02X",
00730                     thischar0[thisrow], thischar1[thisrow],
00731                     thischar2[thisrow], thischar3[thisrow] & 0xFE);
00732             }
00733             else { /* treat as double-width */
00734                 fprintf (outfp,
00735                     "%02X%02X",
00736                     thischar1[thisrow], thischar2[thisrow]);
00737             }
00738         }
00739         fprintf (outfp, "\n");
00740     }
00741 }
00742 }
00743 exit (0);
00744 }

```

5.11.4 Variable Documentation

5.11.4.1 bits_per_pixel

int bits_per_pixel

Definition at line [144](#) 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 [154](#) of file [unibmp2hex.c](#).

5.11.4.4 compression

int compression

Definition at line [145](#) of file [unibmp2hex.c](#).

5.11.4.5 file_size

int file_size

Definition at line [138](#) of file [unibmp2hex.c](#).

5.11.4.6 filetype

char filetype[2]

Definition at line 137 of file [unibmp2hex.c](#).

5.11.4.7 flip

unsigned flip =0

=1 if we're transposing glyph matrix

Definition at line 128 of file [unibmp2hex.c](#).

5.11.4.8 forcewide

unsigned forcewide =0

=1 to set each glyph to 16 pixels wide

Definition at line 129 of file [unibmp2hex.c](#).

5.11.4.9 height

int height

Definition at line 142 of file [unibmp2hex.c](#).

5.11.4.10 hexdigit

unsigned hexdigit[16][4]

32 bit representation of 16x8 0..F bitmap

Definition at line 124 of file [unibmp2hex.c](#).

5.11.4.11 image_offset

int image_offset

Definition at line [139](#) of file [unibmp2hex.c](#).

5.11.4.12 image_size

int image_size

Definition at line [146](#) of file [unibmp2hex.c](#).

5.11.4.13 important_colors

int important_colors

Definition at line [150](#) of file [unibmp2hex.c](#).

5.11.4.14 info_size

int info_size

Definition at line [140](#) of file [unibmp2hex.c](#).

5.11.4.15 ncolors

int ncolors

Definition at line [149](#) of file [unibmp2hex.c](#).

5.11.4.16 nplanes

int nplanes

Definition at line [143](#) of file [unibmp2hex.c](#).

5.11.4.17 planeset

unsigned planeset =0

=1: use plane specified with -p parameter

Definition at line 127 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 132 of file [unibmp2hex.c](#).

5.11.4.19 uniplane

unsigned uniplane =0

Unicode plane number, 0..0xff ff.

Definition at line 126 of file [unibmp2hex.c](#).

5.11.4.20 width

int width

Definition at line 141 of file [unibmp2hex.c](#).

5.11.4.21 x_ppm

int x_ppm

Definition at line 147 of file [unibmp2hex.c](#).

5.11.4.22 y_ppm

int y_ppm

Definition at line 148 of file [unibmp2hex.c](#).

5.12 unibmp2hex.c

[Go to the documentation of this file.](#)

```

00001 /**
00002  @file unibmp2hex.c
00003
00004  @brief unibmp2hex - Turn a .bmp or .wbmp glyph matrix into a
00005           GNU Unifont hex glyph set of 256 characters
00006
00007  @author Paul Hardy, unifoundry <at> unifoundry.com, December 2007
00008
00009  @copyright Copyright (C) 2007, 2008, 2013, 2017, 2019, 2022 Paul Hardy
00010
00011  Synopsis: unibmp2hex [-iin_file.bmp] [-oout_file.hex] [-phex_page_num] [-w]
00012 */
00013 /*
00014
00015  LICENSE:
00016
00017   This program is free software: you can redistribute it and/or modify
00018   it under the terms of the GNU General Public License as published by
00019   the Free Software Foundation, either version 2 of the License, or
00020   (at your option) any later version.
00021
00022   This program is distributed in the hope that it will be useful,
00023   but WITHOUT ANY WARRANTY; without even the implied warranty of
00024   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00025   GNU General Public License for more details.
00026
00027   You should have received a copy of the GNU General Public License
00028   along with this program. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030
00031 /*
00032  20 June 2017 [Paul Hardy]:
00033   - Modify to allow hard-coding of quadruple-width hex glyphs.
00034     The 32nd column (rightmost column) is cleared to zero, because
00035     that column contains the vertical cell border.
00036   - Set U+9FD8..U+9FE9 (complex CJK) to be quadruple-width.
00037   - Set U+011A00..U+011A4F (Masaram Gondi, non-digits) to be wide.
00038   - Set U+011A50..U+011AAF (Soyombo) to be wide.
00039
00040  8 July 2017 [Paul Hardy]:
00041   - All CJK glyphs in the range U+4E00..u+9FFF are double width
00042   again; commented out the line that sets U+9FD8..U+9FE9 to be
00043   quadruple width.
00044
00045  6 August 2017 [Paul Hardy]:
00046   - Remove hard-coding of U+01D200..U+01D24F Ancient Greek Musical
00047     Notation to double-width; allow range to be dual-width.
00048
00049  12 August 2017 [Paul Hardy]:
00050   - Remove Miao script from list of wide scripts, so it can contain
00051     single-width glyphs.
00052
00053  26 December 2017 Paul Hardy:
00054   - Removed Tibetan from list of wide scripts, so it can contain
00055     single-width glyphs.
00056   - Added a number of scripts to be explicitly double-width in case
00057     they are redrawn.
00058   - Added Miao script back as wide, because combining glyphs are
00059     added back to font/plane01/plane01-combining.txt.
00060
00061  05 June 2018 Paul Hardy:
00062   - Made U+2329] and U+232A wide.
00063   - Added to wide settings for CJK Compatibility Forms over entire range.

```



```

00064 - Made Kayah Li script double-width.
00065 - Made U+232A (Right-pointing Angle Bracket) double-width.
00066 - Made U+01F5E7 (Three Rays Right) double-width.
00067
00068 July 2018 Paul Hardy:
00069 - Changed 2017 to 2018 in previous change entry.
00070 - Added Dogra (U+011800..U+01184F) as double width.
00071 - Added Makasar (U+011EE0..U+011EFF) as double width.
00072
00073 23 February 2019 [Paul Hardy]:
00074 - Set U+119A0..U+119FF (Nandinagari) to be wide.
00075 - Set U+1E2C0..U+1E2FF (Wancho) to be wide.
00076
00077 25 May 2019 [Paul Hardy]:
00078 - Added support for the case when the original .bmp monochrome
00079   file has been converted to a 32 bit per pixel RGB file.
00080 - Added support for bitmap images stored from either top to bottom
00081   or bottom to top.
00082 - Add DEBUG compile flag to print header information, to ease
00083   adding support for additional bitmap formats in the future.
00084
00085 6 September 2021 [Paul Hardy]:
00086 - Set U+12F90..U+12FFF (Cypro-Minoan) to be double width.
00087 - Set U+1CF00..U+1CFCF (Znamenny Musical Notation) to be double width.
00088 - Set U+1AFF0..U+1AFFF (Kana Extended-B) to be double width.
00089
00090 13 March 2022 [Paul Hardy]:
00091 - Added support for 24 bits per pixel RGB file.
00092
00093 12 June 2022 [Paul Hardy]:
00094 - Set U+11B00..U+11B5F (Devanagari Extended-A) to be wide.
00095 - Set U+11F00..U+11F5F (Kawi) to be wide.
00096
00097 2 September 2024 [Paul Hardy] - Set these scripts to double width:
00098 - U+10D40..U+10D8F (Garay)
00099 - U+11380..U+113FF (Tulu-Tigalari)
00100 - U+116D0..U+116FF (Myanmar Extended-C)
00101 - U+11F00..U+11F5F (Kawi)
00102 - U+16100..U+1613F (Gurung Khema)
00103 - U+16D40..U+16D7F (Kirat Rai)
00104 - U+18B00..U+18CFF (Khitani Small Script)
00105 - U+1E5D0..U+1E5FF (Ol Onal)
00106
00107 19 April 2025 [Paul Hardy]:
00108 - Remove hard-coding of U+1D100..U+1D1FF (Musical Symbols)
00109   to double-width; allow range to be dual-width.
00110
00111 1 June 2025 [Paul Hardy]:
00112 - Removed Wancho U+1E2C0..U+1E2FF as a wide script; it is now
00113   single-width.
00114 - Added double-width block U+11B60..U+11B7F (Sharada Supplement).
00115 */
00116
00117 #include <stdio.h>
00118 #include <stdlib.h>
00119 #include <string.h>
00120
00121 #define MAXBUF 256 ///< Maximum input file line length - 1
00122
00123
00124 unsigned hexdigit[16][4]; ///< 32 bit representation of 16x8 0..F bitmap
00125
00126 unsigned uniplane=0; ///< Unicode plane number, 0..0xff ff
00127 unsigned planeset=0; ///< =1: use plane specified with -p parameter
00128 unsigned flip=0; ///< =1 if we're transposing glyph matrix
00129 unsigned forcewide=0; ///< =1 to set each glyph to 16 pixels wide
00130
00131 /** The six Unicode plane digits, from left-most (0) to right-most (5) */
00132 unsigned unidigit[6][4];
00133
00134
00135 /** Bitmap Header parameters */
00136 struct {
00137   char filetype[2];
00138   int file_size;
00139   int image_offset;
00140   int info_size;
00141   int width;
00142   int height;
00143   int nplanes;
00144   int bits_per_pixel;

```

```

00145     int compression;
00146     int image_size;
00147     int x_ppm;
00148     int y_ppm;
00149     int ncolors;
00150     int important_colors;
00151 } bmp_header;
00152
00153 /** Bitmap Color Table -- maximum of 256 colors in a BMP file */
00154 unsigned char color_table[256][4]; /* R, G, B, alpha for up to 256 colors */
00155
00156 // #define DEBUG
00157
00158 /**
00159  * @brief The main function.
00160  *
00161  * @param[in] argc The count of command line arguments.
00162  * @param[in] argv Pointer to array of command line arguments.
00163  * @return This program exits with status 0.
00164  */
00165 int
00166 main (int argc, char *argv[])
00167 {
00168     int i, j, k; /* loop variables */
00169     unsigned char inchar; /* temporary input character */
00170     char header[MAXBUF]; /* input buffer for bitmap file header */
00171     int wbmp=0; /* =0 for Windows Bitmap (.bmp); 1 for Wireless Bitmap (.wbmp) */
00172     int fatal; /* =1 if a fatal error occurred */
00173     int match; /* =1 if we're still matching a pattern, 0 if no match */
00174     int empty1, empty2; /* =1 if bytes tested are all zeroes */
00175     unsigned char thischar1[16], thischar2[16]; /* bytes of hex char */
00176     unsigned char thischar0[16], thischar3[16]; /* bytes for quadruple-width */
00177     int thisrow; /* index to point into thischar1[] and thischar2[] */
00178     int tmpsum; /* temporary sum to see if a character is blank */
00179     unsigned this_pixel; /* color of one pixel, if > 1 bit per pixel */
00180     unsigned next_pixels; /* pending group of 8 pixels being read */
00181     unsigned color_mask = 0x00; /* to invert monochrome bitmap, set to 0xFF */
00182
00183     unsigned char bitmap[17*32][18*32/8]; /* final bitmap */
00184     /* For wide array:
00185      * 0 = don't force glyph to double-width;
00186      * 1 = force glyph to double-width;
00187      * 4 = force glyph to quadruple-width.
00188      */
00189     char wide[0x200000]={0x200000 * 0};
00190
00191     char *infile="", *outfile=""; /* names of input and output files */
00192     FILE *infp, *outfp; /* file pointers of input and output files */
00193
00194     if (argc > 1) {
00195         for (i = 1; i < argc; i++) {
00196             if (argv[i][0] == '-') { /* this is an option argument */
00197                 switch (argv[i][1]) {
00198                     case 'i': /* name of input file */
00199                         infile = &argv[i][2];
00200                         break;
00201                     case 'o': /* name of output file */
00202                         outfile = &argv[i][2];
00203                         break;
00204                     case 'p': /* specify a Unicode plane */
00205                         sscanf (&argv[i][2], "%x", &uniplane); /* Get Unicode plane */
00206                         planeset = 1; /* Use specified range, not what's in bitmap */
00207                         break;
00208                     case 'w': /* force wide (16 pixels) for each glyph */
00209                         forcewide = 1;
00210                         break;
00211                     default: /* if unrecognized option, print list and exit */
00212                         fprintf (stderr, "\nSyntax:\n\n");
00213                         fprintf (stderr, " %s -p<Unicode_Page> ", argv[0]);
00214                         fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
00215                         fprintf (stderr, "-w specifies .wbmp output instead of ");
00216                         fprintf (stderr, "default Windows .bmp output.\n\n");
00217                         fprintf (stderr, "-p is followed by 1 to 6 ");
00218                         fprintf (stderr, "Unicode plane hex digits ");
00219                         fprintf (stderr, "(default is Page 0).\n\n");
00220                         fprintf (stderr, "\nExample:\n\n");
00221                         fprintf (stderr, " %s -p83 -iunifont.hex -ou83.bmp\n\n",
00222                             argv[0]);
00223                         exit (1);
00224                 }
00225             }

```

```

00226     }
00227     }
00228 }
00229 /*
00230     Make sure we can open any I/O files that were specified before
00231     doing anything else.
00232 */
00233 if (strlen (infile) > 0) {
00234     if ((infp = fopen (infile, "r")) == NULL) {
00235         fprintf (stderr, "Error: can't open %s for input.\n", infile);
00236         exit (1);
00237     }
00238 }
00239 else {
00240     infp = stdin;
00241 }
00242 if (strlen (outfile) > 0) {
00243     if ((outfp = fopen (outfile, "w")) == NULL) {
00244         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00245         exit (1);
00246     }
00247 }
00248 else {
00249     outfp = stdout;
00250 }
00251 /*
00252     Initialize selected code points for double width (16x16).
00253     Double-width is forced in cases where a glyph (usually a combining
00254     glyph) only occupies the left-hand side of a 16x16 grid, but must
00255     be rendered as double-width to appear properly with other glyphs
00256     in a given script.  If additions were made to a script after
00257     Unicode 5.0, the Unicode version is given in parentheses after
00258     the script name.
00259 */
00260 for (i = 0x0700; i <= 0x074F; i++) wide[i] = 1; /* Syriac */
00261 for (i = 0x0800; i <= 0x083F; i++) wide[i] = 1; /* Samaritan (5.2) */
00262 for (i = 0x0900; i <= 0x0DFF; i++) wide[i] = 1; /* Indic */
00263 for (i = 0x1000; i <= 0x109F; i++) wide[i] = 1; /* Myanmar */
00264 for (i = 0x1100; i <= 0x11FF; i++) wide[i] = 1; /* Hangul Jamo */
00265 for (i = 0x1400; i <= 0x167F; i++) wide[i] = 1; /* Canadian Aboriginal */
00266 for (i = 0x1700; i <= 0x171F; i++) wide[i] = 1; /* Tagalog */
00267 for (i = 0x1720; i <= 0x173F; i++) wide[i] = 1; /* Hanunoo */
00268 for (i = 0x1740; i <= 0x175F; i++) wide[i] = 1; /* Buhid */
00269 for (i = 0x1760; i <= 0x177F; i++) wide[i] = 1; /* Tagbanwa */
00270 for (i = 0x1780; i <= 0x17FF; i++) wide[i] = 1; /* Khmer */
00271 for (i = 0x18B0; i <= 0x18FF; i++) wide[i] = 1; /* Ext. Can. Aboriginal */
00272 for (i = 0x1800; i <= 0x18AF; i++) wide[i] = 1; /* Mongolian */
00273 for (i = 0x1900; i <= 0x194F; i++) wide[i] = 1; /* Limbu */
00274 // for (i = 0x1980; i <= 0x19DF; i++) wide[i] = 1; /* New Tai Lue */
00275 for (i = 0x1A00; i <= 0x1A1F; i++) wide[i] = 1; /* Buginese */
00276 for (i = 0x1A20; i <= 0x1AAF; i++) wide[i] = 1; /* Tai Tham (5.2) */
00277 for (i = 0x1B00; i <= 0x1B7F; i++) wide[i] = 1; /* Balinese */
00278 for (i = 0x1B80; i <= 0x1BBF; i++) wide[i] = 1; /* Sundanese (5.1) */
00279 for (i = 0x1BC0; i <= 0x1BFF; i++) wide[i] = 1; /* Batak (6.0) */
00280 for (i = 0x1C00; i <= 0x1C4F; i++) wide[i] = 1; /* Lepcha (5.1) */
00281 for (i = 0x1CC0; i <= 0x1CCF; i++) wide[i] = 1; /* Sundanese Supplement */
00282 for (i = 0x1CD0; i <= 0x1CFF; i++) wide[i] = 1; /* Vedic Extensions (5.2) */
00283 wide[0x2329] = wide[0x232A] = 1; /* Left- & Right-pointing Angle Brackets */
00284 for (i = 0x2E80; i <= 0xA4CF; i++) wide[i] = 1; /* CJK */
00285 // for (i = 0x9FD8; i <= 0x9FE9; i++) wide[i] = 4; /* CJK quadruple-width */
00286 for (i = 0xA900; i <= 0xA92F; i++) wide[i] = 1; /* Kayah Li (5.1) */
00287 for (i = 0xA930; i <= 0xA95F; i++) wide[i] = 1; /* Rejang (5.1) */
00288 for (i = 0xA960; i <= 0xA97F; i++) wide[i] = 1; /* Hangul Jamo Extended-A */
00289 for (i = 0xA980; i <= 0xA9DF; i++) wide[i] = 1; /* Javanese (5.2) */
00290 for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham (5.1) */
00291 for (i = 0xA9E0; i <= 0xA9FF; i++) wide[i] = 1; /* Myanmar Extended-B */
00292 for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham */
00293 for (i = 0xAA60; i <= 0xAA7F; i++) wide[i] = 1; /* Myanmar Extended-A */
00294 for (i = 0xAAE0; i <= 0xA AFF; i++) wide[i] = 1; /* Meetei Mayek Ext (6.0) */
00295 for (i = 0xABC0; i <= 0xABFF; i++) wide[i] = 1; /* Meetei Mayek (5.2) */
00296 for (i = 0xAC00; i <= 0xD7AF; i++) wide[i] = 1; /* Hangul Syllables */
00297 for (i = 0xD7B0; i <= 0xD7FF; i++) wide[i] = 1; /* Hangul Jamo Extended-B */
00298 for (i = 0xF900; i <= 0xFAFF; i++) wide[i] = 1; /* CJK Compatibility */
00299 for (i = 0xFE10; i <= 0xFE1F; i++) wide[i] = 1; /* Vertical Forms */
00300 for (i = 0xFE30; i <= 0xFE60; i++) wide[i] = 1; /* CJK Compatibility Forms */
00301 for (i = 0xFFE0; i <= 0xFFE6; i++) wide[i] = 1; /* CJK Compatibility Forms */
00302
00303 wide[0x303F] = 0; /* CJK half-space fill */
00304
00305 /* Supplemental Multilingual Plane (Plane 01) */
00306 for (i = 0x0105C0; i <= 0x0105FF; i++) wide[i] = 1; /* Toghri */

```

```

00307 for (i = 0x010A00; i <= 0x010A5F; i++) wide[i] = 1; /* Kharoshthi */
00308 for (i = 0x011000; i <= 0x01107F; i++) wide[i] = 1; /* Brahmi */
00309 for (i = 0x011080; i <= 0x0110CF; i++) wide[i] = 1; /* Kaithi */
00310 for (i = 0x011100; i <= 0x01114F; i++) wide[i] = 1; /* Chakma */
00311 for (i = 0x011180; i <= 0x0111DF; i++) wide[i] = 1; /* Sharada */
00312 for (i = 0x011200; i <= 0x01124F; i++) wide[i] = 1; /* Khojki */
00313 for (i = 0x0112B0; i <= 0x0112FF; i++) wide[i] = 1; /* Khudawadi */
00314 for (i = 0x011300; i <= 0x01137F; i++) wide[i] = 1; /* Grantha */
00315 for (i = 0x011380; i <= 0x0113FF; i++) wide[i] = 1; /* Tulu-Tigalari */
00316 for (i = 0x011400; i <= 0x01147F; i++) wide[i] = 1; /* Newa */
00317 for (i = 0x011480; i <= 0x0114DF; i++) wide[i] = 1; /* Tirhuta */
00318 for (i = 0x011580; i <= 0x0115FF; i++) wide[i] = 1; /* Siddham */
00319 for (i = 0x011600; i <= 0x01165F; i++) wide[i] = 1; /* Modi */
00320 for (i = 0x011660; i <= 0x01167F; i++) wide[i] = 1; /* Mongolian Suppl. */
00321 for (i = 0x011680; i <= 0x0116CF; i++) wide[i] = 1; /* Takri */
00322 for (i = 0x0116D0; i <= 0x0116FF; i++) wide[i] = 1; /* Myanmar Extended-C */
00323 for (i = 0x011700; i <= 0x01173F; i++) wide[i] = 1; /* Ahom */
00324 for (i = 0x011800; i <= 0x01184F; i++) wide[i] = 1; /* Dogra */
00325 for (i = 0x011900; i <= 0x01195F; i++) wide[i] = 1; /* Dives Akuru */
00326 for (i = 0x0119A0; i <= 0x0119FF; i++) wide[i] = 1; /* Nandinagari */
00327 for (i = 0x011A00; i <= 0x011A4F; i++) wide[i] = 1; /* Zanabazar Square */
00328 for (i = 0x011A50; i <= 0x011AAF; i++) wide[i] = 1; /* Soyombo */
00329 for (i = 0x011B00; i <= 0x011B5F; i++) wide[i] = 1; /* Devanagari Extended-A */
00330 for (i = 0x011B60; i <= 0x011B7F; i++) wide[i] = 1; /* Sharada Supplement */
00331 for (i = 0x011F00; i <= 0x011F5F; i++) wide[i] = 1; /* Kawi */
00332 for (i = 0x011C00; i <= 0x011C6F; i++) wide[i] = 1; /* Bhaiksuki */
00333 for (i = 0x011C70; i <= 0x011CBF; i++) wide[i] = 1; /* Marchen */
00334 for (i = 0x011D00; i <= 0x011D5F; i++) wide[i] = 1; /* Masaram Gondi */
00335 for (i = 0x011EE0; i <= 0x011EFF; i++) wide[i] = 1; /* Makasar */
00336 for (i = 0x011F00; i <= 0x011F5F; i++) wide[i] = 1; /* Kawi */
00337 for (i = 0x012F90; i <= 0x012FFF; i++) wide[i] = 1; /* Cypro-Minoan */
00338 /* Make Bassa Vah all single width or all double width */
00339 for (i = 0x016100; i <= 0x01613F; i++) wide[i] = 1; /* Gurung Khema */
00340 for (i = 0x016AD0; i <= 0x016AFF; i++) wide[i] = 1; /* Bassa Vah */
00341 for (i = 0x016B00; i <= 0x016B8F; i++) wide[i] = 1; /* Pahawh Hmong */
00342 for (i = 0x016D40; i <= 0x016D7F; i++) wide[i] = 1; /* Kirat Rai */
00343 for (i = 0x016F00; i <= 0x016F9F; i++) wide[i] = 1; /* Miao */
00344 for (i = 0x016FE0; i <= 0x016FFF; i++) wide[i] = 1; /* Ideograph Sym/Punct */
00345 for (i = 0x017000; i <= 0x0187FF; i++) wide[i] = 1; /* Tangut */
00346 for (i = 0x018800; i <= 0x018AFF; i++) wide[i] = 1; /* Tangut Components */
00347 for (i = 0x018B00; i <= 0x018CFF; i++) wide[i] = 1; /* Khitan Small Script */
00348 for (i = 0x01AFF0; i <= 0x01AFFF; i++) wide[i] = 1; /* Kana Extended-B */
00349 for (i = 0x01B000; i <= 0x01B0FF; i++) wide[i] = 1; /* Kana Supplement */
00350 for (i = 0x01B100; i <= 0x01B12F; i++) wide[i] = 1; /* Kana Extended-A */
00351 for (i = 0x01B170; i <= 0x01B2FF; i++) wide[i] = 1; /* Nushu */
00352 for (i = 0x01CF00; i <= 0x01CFCF; i++) wide[i] = 1; /* Znamenny Musical */
00353 for (i = 0x01D800; i <= 0x01DAAF; i++) wide[i] = 1; /* Sutton SignWriting */
00354 for (i = 0x01E500; i <= 0x01E5FF; i++) wide[i] = 1; /* Ol Onal */
00355 for (i = 0x01E800; i <= 0x01E8DF; i++) wide[i] = 1; /* Mende Kikakui */
00356 for (i = 0x01F200; i <= 0x01F2FF; i++) wide[i] = 1; /* Encl Ideograp Suppl */
00357 wide[0x01F5E7] = 1; /* Three Rays Right */
00358
00359 /*
00360 Determine whether or not the file is a Microsoft Windows Bitmap file.
00361 If it starts with 'B', 'M', assume it's a Windows Bitmap file.
00362 Otherwise, assume it's a Wireless Bitmap file.
00363
00364 WARNING: There isn't much in the way of error checking here --
00365 if you give it a file that wasn't first created by hex2bmp.c,
00366 all bets are off.
00367 */
00368 fatal = 0; /* assume everything is okay with reading input file */
00369 if ((header[0] = fgetc (infp)) != EOF) {
00370     if ((header[1] = fgetc (infp)) != EOF) {
00371         if (header[0] == 'B' && header[1] == 'M') {
00372             wbmp = 0; /* Not a Wireless Bitmap -- it's a Windows Bitmap */
00373         }
00374         else {
00375             wbmp = 1; /* Assume it's a Wireless Bitmap */
00376         }
00377     }
00378     else
00379         fatal = 1;
00380 }
00381 else
00382     fatal = 1;
00383
00384 if (fatal) {
00385     fprintf (stderr, "Fatal error; end of input file.\n\n");
00386     exit (1);
00387 }

```

```

00388  /*
00389   If this is a Wireless Bitmap (.wbmp) format file,
00390   skip the header and point to the start of the bitmap itself.
00391  */
00392  if (wbmp) {
00393      for (i=2; i<6; i++)
00394          header[i] = fgetc (infp);
00395      /*
00396       Now read the bitmap.
00397      */
00398      for (i=0; i < 32*17; i++) {
00399          for (j=0; j < 32*18/8; j++) {
00400              inchar = fgetc (infp);
00401              bitmap[i][j] = ~inchar; /* invert bits for proper color */
00402          }
00403      }
00404  }
00405  /*
00406   Otherwise, treat this as a Windows Bitmap file, because we checked
00407   that it began with "BM". Save the header contents for future use.
00408   Expect a 14 byte standard BITMAPFILEHEADER format header followed
00409   by a 40 byte standard BITMAPINFOHEADER Device Independent Bitmap
00410   header, with data stored in little-endian format.
00411  */
00412  else {
00413      for (i = 2; i < 54; i++)
00414          header[i] = fgetc (infp);
00415
00416      bmp_header.filetype[0] = 'B';
00417      bmp_header.filetype[1] = 'M';
00418
00419      bmp_header.file_size =
00420          (header[2] & 0xFF) | ((header[3] & 0xFF) << 8) |
00421          ((header[4] & 0xFF) << 16) | ((header[5] & 0xFF) << 24);
00422
00423      /* header bytes 6..9 are reserved */
00424
00425      bmp_header.image_offset =
00426          (header[10] & 0xFF) | ((header[11] & 0xFF) << 8) |
00427          ((header[12] & 0xFF) << 16) | ((header[13] & 0xFF) << 24);
00428
00429      bmp_header.info_size =
00430          (header[14] & 0xFF) | ((header[15] & 0xFF) << 8) |
00431          ((header[16] & 0xFF) << 16) | ((header[17] & 0xFF) << 24);
00432
00433      bmp_header.width =
00434          (header[18] & 0xFF) | ((header[19] & 0xFF) << 8) |
00435          ((header[20] & 0xFF) << 16) | ((header[21] & 0xFF) << 24);
00436
00437      bmp_header.height =
00438          (header[22] & 0xFF) | ((header[23] & 0xFF) << 8) |
00439          ((header[24] & 0xFF) << 16) | ((header[25] & 0xFF) << 24);
00440
00441      bmp_header.nplanes =
00442          (header[26] & 0xFF) | ((header[27] & 0xFF) << 8);
00443
00444      bmp_header.bits_per_pixel =
00445          (header[28] & 0xFF) | ((header[29] & 0xFF) << 8);
00446
00447      bmp_header.compression =
00448          (header[30] & 0xFF) | ((header[31] & 0xFF) << 8) |
00449          ((header[32] & 0xFF) << 16) | ((header[33] & 0xFF) << 24);
00450
00451      bmp_header.image_size =
00452          (header[34] & 0xFF) | ((header[35] & 0xFF) << 8) |
00453          ((header[36] & 0xFF) << 16) | ((header[37] & 0xFF) << 24);
00454
00455      bmp_header.x_ppm =
00456          (header[38] & 0xFF) | ((header[39] & 0xFF) << 8) |
00457          ((header[40] & 0xFF) << 16) | ((header[41] & 0xFF) << 24);
00458
00459      bmp_header.y_ppm =
00460          (header[42] & 0xFF) | ((header[43] & 0xFF) << 8) |
00461          ((header[44] & 0xFF) << 16) | ((header[45] & 0xFF) << 24);
00462
00463      bmp_header.ncolors =
00464          (header[46] & 0xFF) | ((header[47] & 0xFF) << 8) |
00465          ((header[48] & 0xFF) << 16) | ((header[49] & 0xFF) << 24);
00466
00467      bmp_header.important_colors =
00468          (header[50] & 0xFF) | ((header[51] & 0xFF) << 8) |

```

```

00469         ((header[52] & 0xFF) « 16) | ((header[53] & 0xFF) « 24);
00470
00471     if (bmp_header.ncolors == 0)
00472         bmp_header.ncolors = 1 « bmp_header.bits_per_pixel;
00473
00474     /* If a Color Table exists, read it */
00475     if (bmp_header.ncolors > 0 && bmp_header.bits_per_pixel <= 8) {
00476         for (i = 0; i < bmp_header.ncolors; i++) {
00477             color_table[i][0] = fgetc (infp); /* Red */
00478             color_table[i][1] = fgetc (infp); /* Green */
00479             color_table[i][2] = fgetc (infp); /* Blue */
00480             color_table[i][3] = fgetc (infp); /* Alpha */
00481         }
00482     /*
00483      * Determine from the first color table entry whether we
00484      * are inverting the resulting bitmap image.
00485      */
00486     if ( (color_table[0][0] + color_table[0][1] + color_table[0][2])
00487          < (3 * 128) ) {
00488         color_mask = 0xFF;
00489     }
00490 }
00491
00492 #ifdef DEBUG
00493
00494     /*
00495     * Print header info for possibly adding support for
00496     * additional file formats in the future, to determine
00497     * how the bitmap is encoded.
00498     */
00499     fprintf (stderr, "Filetype: '%c%c'\n",
00500             bmp_header.filetype[0], bmp_header.filetype[1]);
00501     fprintf (stderr, "File Size: %d\n", bmp_header.file_size);
00502     fprintf (stderr, "Image Offset: %d\n", bmp_header.image_offset);
00503     fprintf (stderr, "Info Header Size: %d\n", bmp_header.info_size);
00504     fprintf (stderr, "Image Width: %d\n", bmp_header.width);
00505     fprintf (stderr, "Image Height: %d\n", bmp_header.height);
00506     fprintf (stderr, "Number of Planes: %d\n", bmp_header.nplanes);
00507     fprintf (stderr, "Bits per Pixel: %d\n", bmp_header.bits_per_pixel);
00508     fprintf (stderr, "Compression Method: %d\n", bmp_header.compression);
00509     fprintf (stderr, "Image Size: %d\n", bmp_header.image_size);
00510     fprintf (stderr, "X Pixels per Meter: %d\n", bmp_header.x_ppm);
00511     fprintf (stderr, "Y Pixels per Meter: %d\n", bmp_header.y_ppm);
00512     fprintf (stderr, "Number of Colors: %d\n", bmp_header.ncolors);
00513     fprintf (stderr, "Important Colors: %d\n", bmp_header.important_colors);
00514
00515 #endif
00516
00517     /*
00518     * Now read the bitmap.
00519     */
00520     for (i = 32*17-1; i >= 0; i--) {
00521         for (j=0; j < 32*18/8; j++) {
00522             next_pixels = 0x00; /* initialize next group of 8 pixels */
00523             /* Read a monochrome image -- the original case */
00524             if (bmp_header.bits_per_pixel == 1) {
00525                 next_pixels = fgetc (infp);
00526             }
00527             /* Read a 32 bit per pixel RGB image; convert to monochrome */
00528             else if ( bmp_header.bits_per_pixel == 24 ||
00529                      bmp_header.bits_per_pixel == 32) {
00530                 next_pixels = 0;
00531                 for (k = 0; k < 8; k++) { /* get next 8 pixels */
00532                     this_pixel = (fgetc (infp) & 0xFF) +
00533                                 (fgetc (infp) & 0xFF) +
00534                                 (fgetc (infp) & 0xFF);
00535
00536                     if (bmp_header.bits_per_pixel == 32) {
00537                         (void) fgetc (infp); /* ignore alpha value */
00538                     }
00539
00540                     /* convert RGB color space to monochrome */
00541                     if (this_pixel >= (128 * 3))
00542                         this_pixel = 0;
00543                     else
00544                         this_pixel = 1;
00545
00546                     /* shift next pixel color into place for 8 pixels total */
00547                     next_pixels = (next_pixels « 1) | this_pixel;
00548                 }
00549             }

```

```

00550         if (bmp_header.height < 0) { /* Bitmap drawn top to bottom */
00551             bitmap [(32*17-1) - i][j] = next_pixels;
00552         }
00553         else { /* Bitmap drawn bottom to top */
00554             bitmap [i][j] = next_pixels;
00555         }
00556     }
00557 }
00558
00559 /*
00560  If any bits are set in color_mask, apply it to
00561  entire bitmap to invert black <--> white.
00562 */
00563 if (color_mask != 0x00) {
00564     for (i = 32*17-1; i >= 0; i--) {
00565         for (j=0; j < 32*18/8; j++) {
00566             bitmap [i][j] ^= color_mask;
00567         }
00568     }
00569 }
00570
00571 }
00572
00573 /*
00574  We've read the entire file. Now close the input file pointer.
00575 */
00576 fclose (infp);
00577 /*
00578  We now have the header portion in the header[] array,
00579  and have the bitmap portion from top-to-bottom in the bitmap[] array.
00580 */
00581 /*
00582  If no Unicode range (U+nnnnnn00 through U+nnnnnnFF) was specified
00583  with a -p parameter, determine the range from the digits in the
00584  bitmap itself.
00585
00586  Store bitmaps for the hex digit patterns that this file uses.
00587 */
00588 if (!planeset) { /* If Unicode range not specified with -p parameter */
00589     for (i = 0x0; i <= 0xF; i++) { /* hex digit pattern we're storing */
00590         for (j = 0; j < 4; j++) {
00591             hexdigit[i][j] =
00592                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8][6] << 24) |
00593                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 1][6] << 16) |
00594                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 2][6] << 8) |
00595                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 3][6]);
00596         }
00597     }
00598     /*
00599      Read the Unicode plane digits into arrays for comparison, to
00600      determine the upper four hex digits of the glyph addresses.
00601     */
00602     for (i = 0; i < 4; i++) {
00603         for (j = 0; j < 4; j++) {
00604             unidigit[i][j] =
00605                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 1][i + 3] << 24) |
00606                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 2][i + 3] << 16) |
00607                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 3][i + 3] << 8) |
00608                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 4][i + 3]);
00609         }
00610     }
00611
00612     tmpsum = 0;
00613     for (i = 4; i < 6; i++) {
00614         for (j = 0; j < 4; j++) {
00615             unidigit[i][j] =
00616                 ((unsigned)bitmap[32 * 1 + 4 * j + 8][i] << 24) |
00617                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 1][i] << 16) |
00618                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 2][i] << 8) |
00619                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 3][i]);
00620             tmpsum |= unidigit[i][j];
00621         }
00622     }
00623     if (tmpsum == 0) { /* the glyph matrix is transposed */
00624         flip = 1; /* note transposed order for processing glyphs in matrix */
00625         /*
00626          Get 5th and 6th hex digits by shifting first column header left by
00627          1.5 columns, thereby shifting the hex digit right after the leading
00628          "U+nnnn" page number.
00629         */
00630         for (i = 0x08; i < 0x18; i++) {

```



```

00631     bitmap[i][7] = (bitmap[i][8] << 4) | ((bitmap[i][ 9] >> 4) & 0xf);
00632     bitmap[i][8] = (bitmap[i][9] << 4) | ((bitmap[i][10] >> 4) & 0xf);
00633 }
00634 for (i = 4; i < 6; i++) {
00635     for (j = 0; j < 4; j++) {
00636         unidigit[i][j] =
00637             ((unsigned)bitmap[4 * j + 8 + 1][i + 3] << 24) |
00638             ((unsigned)bitmap[4 * j + 8 + 2][i + 3] << 16) |
00639             ((unsigned)bitmap[4 * j + 8 + 3][i + 3] << 8) |
00640             ((unsigned)bitmap[4 * j + 8 + 4][i + 3]);
00641     }
00642 }
00643 }
00644
00645 /*
00646     Now determine the Unicode plane by comparing unidigit[0..5] to
00647     the hexdigit[0x0..0xF] array.
00648 */
00649 uniplane = 0;
00650 for (i=0; i<6; i++) { /* go through one bitmap digit at a time */
00651     match = 0; /* haven't found pattern yet */
00652     for (j = 0x0; !match && j <= 0xF; j++) {
00653         if (unidigit[i][0] == hexdigit[j][0] &&
00654             unidigit[i][1] == hexdigit[j][1] &&
00655             unidigit[i][2] == hexdigit[j][2] &&
00656             unidigit[i][3] == hexdigit[j][3]) { /* we found the digit */
00657             uniplane |= j;
00658             match = 1;
00659         }
00660     }
00661     uniplane <= 4;
00662 }
00663 uniplane >= 4;
00664 }
00665 /*
00666     Now read each glyph and print it as hex.
00667 */
00668 for (i = 0x0; i <= 0xf; i++) {
00669     for (j = 0x0; j <= 0xf; j++) {
00670         for (k = 0; k < 16; k++) {
00671             if (flip) { /* transpose glyph matrix */
00672                 thischar0[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) ];
00673                 thischar1[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 1];
00674                 thischar2[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 2];
00675                 thischar3[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 3];
00676             }
00677             else {
00678                 thischar0[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) ];
00679                 thischar1[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 1];
00680                 thischar2[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 2];
00681                 thischar3[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 3];
00682             }
00683         }
00684     }
00685     /*
00686         If the second half of the 16*16 character is all zeroes, this
00687         character is only 8 bits wide, so print a half-width character.
00688     */
00689     empty1 = empty2 = 1;
00690     for (k=0; (empty1 || empty2) && k < 16; k++) {
00691         if (thischar1[k] != 0) empty1 = 0;
00692         if (thischar2[k] != 0) empty2 = 0;
00693     }
00694     /*
00695         Only print this glyph if it isn't blank.
00696     */
00697     if (!empty1 || !empty2) {
00698         /*
00699             If the second half is empty, this is a half-width character.
00700             Only print the first half.
00701         */
00702         /*
00703             Original GNU Unifont format is four hexadecimal digit character
00704             code followed by a colon followed by a hex string. Add support
00705             for codes beyond the Basic Multilingual Plane.
00706
00707             Unicode ranges from U+0000 to U+10FFFF, so print either a
00708             4-digit or a 6-digit code point. Note that this software
00709             should support up to an 8-digit code point, extending beyond
00710             the normal Unicode range, but this has not been fully tested.
00711         */
00712         if (uniplane > 0xff)

```



```

00712         fprintf (outfp, "%04X%X%X:", uniplane, i, j); // 6 digit code pt.
00713     else
00714         fprintf (outfp, "%02X%X%X:", uniplane, i, j); // 4 digit code pt.
00715     for (thisrow=0; thisrow<16; thisrow++) {
00716         /*
00717          * If second half is empty and we're not forcing this
00718          * code point to double width, print as single width.
00719          */
00720         if (!forcewide &&
00721             empty2 && !wide[(uniplane « 8) | (i « 4) | j]) {
00722             fprintf (outfp,
00723                     "%02X",
00724                     thischar1[thisrow]);
00725         }
00726         else if (wide[(uniplane « 8) | (i « 4) | j] == 4) {
00727             /* quadruple-width; force 32nd pixel to zero */
00728             fprintf (outfp,
00729                     "%02X%02X%02X%02X",
00730                     thischar0[thisrow], thischar1[thisrow],
00731                     thischar2[thisrow], thischar3[thisrow] & 0xFE);
00732         }
00733         else { /* treat as double-width */
00734             fprintf (outfp,
00735                     "%02X%02X",
00736                     thischar1[thisrow], thischar2[thisrow]);
00737         }
00738     }
00739     fprintf (outfp, "\n");
00740 }
00741 }
00742 }
00743 exit (0);
00744 }

```

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 */
```

```

00058 unsigned image_start;    /* byte offset of image in file */
00059
00060 /*
00061  Values preserved from Device Independent Bitmap (DIB) Header.
00062
00063  The DIB fields below are in the standard 40-byte header.  Version
00064  4 and version 5 headers have more information, mainly for color
00065  information.  That is skipped over, because a valid glyph image
00066  is just monochrome.
00067 */
00068 int dib_length;           /* in bytes, for parsing by header version */
00069 int image_width = 0;      /* Signed image width */
00070 int image_height = 0;     /* Signed image height */
00071 int num_planes = 0;       /* number of planes; must be 1 */
00072 int bits_per_pixel = 0;   /* for palletized color maps (< 2^16 colors) */
00073 /*
00074  The following fields are not in the original spec, so initialize
00075  them to 0 so we can correctly parse an original file format.
00076 */
00077 int compression_method=0; /* 0 --> uncompressed RGB/monochrome */
00078 int image_size = 0;       /* 0 is a valid size if no compression */
00079 int hres = 0;             /* image horizontal resolution */
00080 int vres = 0;            /* image vertical resolution */
00081 int num_colors = 0;       /* Number of colors for palletized images */
00082 int important_colors = 0; /* Number of significant colors (0 or 2) */
00083
00084 int true_colors = 0;      /* interpret num_colors, which can equal 0 */
00085
00086 /*
00087  Color map.  This should be a monochrome file, so only two
00088  colors are stored.
00089 */
00090 unsigned char color_map[2][4]; /* two of R, G, B, and possibly alpha */
00091
00092 /*
00093  The monochrome image bitmap, stored as a vector 544 rows by
00094  72*8 columns.
00095 */
00096 unsigned image_bytes[544*72];
00097
00098 /*
00099  Flags for conversion & I/O.
00100 */
00101 int verbose = 0;          /* Whether to print file info on stderr */
00102 unsigned image_xor = 0x00; /* Invert (= 0xFF) if color 0 is not black */
00103
00104 /*
00105  Temporary variables.
00106 */
00107 int i, j, k;              /* loop variables */
00108
00109 /* Compression type, for parsing file */
00110 char *compression_type[MAX_COMPRESSION_METHOD + 1] = {
00111     "BI_RGB",             /* 0 */
00112     "BI_RLE8",            /* 1 */
00113     "BI_RLE4",            /* 2 */
00114     "BI_BITFIELDS",       /* 3 */
00115     "BI_JPEG",            /* 4 */
00116     "BI_PNG",             /* 5 */
00117     "BI_ALPHABITFIELDS", /* 6 */
00118     "", "", "", "",       /* 7 - 10 */
00119     "BI_CMYK",            /* 11 */
00120     "BI_CMYKRLE8",        /* 12 */
00121     "BI_CMYKRLE4",        /* 13 */
00122 };
00123
00124 /* Standard unihex2bmp.c header for BMP image */
00125 unsigned standard_header[62] = {
00126     /* 0 */ 0x42, 0x4d, 0x3e, 0x99, 0x00, 0x00, 0x00, 0x00,
00127     /* 8 */ 0x00, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x28, 0x00,
00128     /* 16 */ 0x00, 0x00, 0x40, 0x02, 0x00, 0x00, 0x20, 0x02,
00129     /* 24 */ 0x00, 0x00, 0x01, 0x00, 0x01, 0x00, 0x00, 0x00,
00130     /* 32 */ 0x00, 0x00, 0x00, 0x99, 0x00, 0x00, 0xc4, 0x0e,
00131     /* 40 */ 0x00, 0x00, 0xc4, 0x0e, 0x00, 0x00, 0x00, 0x00,
00132     /* 48 */ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
00133     /* 56 */ 0x00, 0x00, 0xff, 0xff, 0xff, 0x00,
00134 };
00135
00136 unsigned get_bytes (FILE *, int);
00137 void regrid (unsigned *);
00138

```

```

00139 char *infile="", *outfile=""; /* names of input and output files */
00140 FILE *infp, *outfp; /* file pointers of input and output files */
00141
00142 /*
00143  Process command line arguments.
00144 */
00145 if (argc > 1) {
00146     for (i = 1; i < argc; i++) {
00147         if (argv[i][0] == '-') { /* this is an option argument */
00148             switch (argv[i][1]) {
00149                 case 'i': /* name of input file */
00150                     infile = &argv[i][2];
00151                     break;
00152                 case 'o': /* name of output file */
00153                     outfile = &argv[i][2];
00154                     break;
00155                 case 'v': /* verbose output */
00156                     verbose = 1;
00157                     break;
00158                 case 'V': /* print version & quit */
00159                     fprintf(stderr, "unibmpbump version %s\n", VERSION);
00160                     exit (EXIT_SUCCESS);
00161                     break;
00162                 case '-': /* see if "--verbose" */
00163                     if (strcmp (argv[i], "--verbose") == 0) {
00164                         verbose = 1;
00165                     }
00166                     else if (strcmp (argv[i], "--version") == 0) {
00167                         fprintf (stderr, "unibmpbump version %s\n", VERSION);
00168                         exit (EXIT_SUCCESS);
00169                     }
00170                     break;
00171                 default: /* if unrecognized option, print list and exit */
00172                     fprintf (stderr, "\nSyntax:\n");
00173                     fprintf (stderr, "    unibmpbump ");
00174                     fprintf (stderr, "-i<Input_File> -o<Output_File>\n");
00175                     fprintf (stderr, "-v or --verbose gives verbose output");
00176                     fprintf (stderr, " on stderr\n");
00177                     fprintf (stderr, "-V or --version prints version");
00178                     fprintf (stderr, " on stderr and exits\n");
00179                     fprintf (stderr, "\nExample:\n");
00180                     fprintf (stderr, "    unibmpbump -iuni0101.bmp");
00181                     fprintf (stderr, " -onew-uni0101.bmp\n");
00182                     exit (EXIT_SUCCESS);
00183             }
00184         }
00185     }
00186 }
00187
00188 /*
00189  Make sure we can open any I/O files that were specified before
00190  doing anything else.
00191 */
00192 if (strlen (infile) > 0) {
00193     if ((infp = fopen (infile, "r")) == NULL) {
00194         fprintf (stderr, "Error: can't open %s for input.\n", infile);
00195         exit (EXIT_FAILURE);
00196     }
00197 }
00198 else {
00199     infp = stdin;
00200 }
00201 if (strlen (outfile) > 0) {
00202     if ((outfp = fopen (outfile, "w")) == NULL) {
00203         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00204         exit (EXIT_FAILURE);
00205     }
00206 }
00207 else {
00208     outfp = stdout;
00209 }
00210
00211 /* Read bitmap file header */
00212 file_format[0] = get_bytes (infp, 1);
00213 file_format[1] = get_bytes (infp, 1);
00214 file_format[2] = '\0'; /* Terminate string with null */
00215
00216 /* Read file size */
00217 filesize = get_bytes (infp, 4);
00218
00219

```

```

00220  /* Read Reserved bytes */
00221  rsvd_hdr[0] = get_bytes (infp, 1);
00222  rsvd_hdr[1] = get_bytes (infp, 1);
00223  rsvd_hdr[2] = get_bytes (infp, 1);
00224  rsvd_hdr[3] = get_bytes (infp, 1);
00225
00226  /* Read Image Offset Address within file */
00227  image_start = get_bytes (infp, 4);
00228
00229  /*
00230   See if this looks like a valid image file based on
00231   the file header first two bytes.
00232  */
00233  if (strncmp (file_format, "BM", 2) != 0) {
00234      fprintf (stderr, "\nInvalid file format: not file type \"BM\".\n\n");
00235      exit (EXIT_FAILURE);
00236  }
00237
00238  if (verbose) {
00239      fprintf (stderr, "\nFile Header:\n");
00240      fprintf (stderr, "  File Type:  \"%s\"\n", file_format);
00241      fprintf (stderr, "  File Size:  %d bytes\n", filesize);
00242      fprintf (stderr, "  Reserved:  ");
00243      for (i = 0; i < 4; i++) fprintf (stderr, " 0x%02X", rsvd_hdr[i]);
00244      fputc ('\n', stderr);
00245      fprintf (stderr, "  Image Start: %d. = 0x%02X = 0%05o\n\n",
00246              image_start, image_start, image_start);
00247  } /* if (verbose) */
00248
00249  /*
00250   Device Independent Bitmap (DIB) Header: bitmap information header
00251   ("BM" format file DIB Header is 12 bytes long).
00252  */
00253  dib_length = get_bytes (infp, 4);
00254
00255  /*
00256   Parse one of three versions of Device Independent Bitmap (DIB) format:
00257
00258   Length  Format
00259   -----
00260   12     BITMAPCOREHEADER
00261   40     BITMAPINFOHEADER
00262   108    BITMAPV4HEADER
00263   124    BITMAPV5HEADER
00264  */
00265  if (dib_length == 12) { /* BITMAPCOREHEADER format -- UNTESTED */
00266      image_width  = get_bytes (infp, 2);
00267      image_height = get_bytes (infp, 2);
00268      num_planes   = get_bytes (infp, 2);
00269      bits_per_pixel = get_bytes (infp, 2);
00270  }
00271  else if (dib_length >= 40) { /* BITMAPINFOHEADER format or later */
00272      image_width  = get_bytes (infp, 4);
00273      image_height = get_bytes (infp, 4);
00274      num_planes   = get_bytes (infp, 2);
00275      bits_per_pixel = get_bytes (infp, 2);
00276      compression_method = get_bytes (infp, 4); /* BI_BITFIELDS */
00277      image_size    = get_bytes (infp, 4);
00278      hres          = get_bytes (infp, 4);
00279      vres          = get_bytes (infp, 4);
00280      num_colors    = get_bytes (infp, 4);
00281      important_colors = get_bytes (infp, 4);
00282
00283      /* true_colors is true number of colors in image */
00284      if (num_colors == 0)
00285          true_colors = 1 « bits_per_pixel;
00286      else
00287          true_colors = num_colors;
00288
00289      /*
00290       If dib_length > 40, the format is BITMAPV4HEADER or
00291       BITMAPV5HEADER. As this program is only designed
00292       to handle a monochrome image, we can ignore the rest
00293       of the header but must read past the remaining bytes.
00294      */
00295      for (i = 40; i < dib_length; i++) (void) get_bytes (infp, 1);
00296  }
00297
00298  if (verbose) {
00299      fprintf (stderr, "Device Independent Bitmap (DIB) Header:\n");
00300      fprintf (stderr, "  DIB Length:  %9d bytes (version = ", dib_length);

```

```

00301
00302     if (dib_length == 12) fprintf(stderr, "\\\"BITMAPCOREHEADER\\\"\n");
00303     else if (dib_length == 40) fprintf(stderr, "\\\"BITMAPINFOHEADER\\\"\n");
00304     else if (dib_length == 108) fprintf(stderr, "\\\"BITMAPV4HEADER\\\"\n");
00305     else if (dib_length == 124) fprintf(stderr, "\\\"BITMAPV5HEADER\\\"\n");
00306     else fprintf(stderr, "unknown");
00307     fprintf(stderr, "    Bitmap Width:   %6d pixels\n", image_width);
00308     fprintf(stderr, "    Bitmap Height:  %6d pixels\n", image_height);
00309     fprintf(stderr, "    Color Planes:   %6d\n",    num_planes);
00310     fprintf(stderr, "    Bits per Pixel: %6d\n",    bits_per_pixel);
00311     fprintf(stderr, "    Compression Method: %2d --> ", compression_method);
00312     if (compression_method <= MAX_COMPRESSION_METHOD) {
00313         fprintf(stderr, "%s", compression_type [compression_method]);
00314     }
00315     /*
00316     Supported compression method values:
00317         0 --> uncompressed RGB
00318         11 --> uncompressed CMYK
00319     */
00320     if (compression_method == 0 || compression_method == 11) {
00321         fprintf(stderr, " (no compression)");
00322     }
00323     else {
00324         fprintf(stderr, "Image uses compression; this is unsupported.\n\n");
00325         exit (EXIT_FAILURE);
00326     }
00327     fprintf(stderr, "\\n");
00328     fprintf(stderr, "    Image Size:           %5d bytes\n", image_size);
00329     fprintf(stderr, "    Horizontal Resolution: %5d pixels/meter\n", hres);
00330     fprintf(stderr, "    Vertical Resolution:   %5d pixels/meter\n", vres);
00331     fprintf(stderr, "    Number of Colors:      %5d", num_colors);
00332     if (num_colors != true_colors) {
00333         fprintf(stderr, " --> %d", true_colors);
00334     }
00335     fputc ('\n', stderr);
00336     fprintf(stderr, "    Important Colors:      %5d", important_colors);
00337     if (important_colors == 0)
00338         fprintf(stderr, " (all colors are important)");
00339     fprintf(stderr, "\\n\n");
00340 } /* if (verbose) */
00341
00342 /*
00343 Print Color Table information for images with pallettized colors.
00344 */
00345 if (bits_per_pixel <= 8) {
00346     for (i = 0; i < 2; i++) {
00347         color_map [i][0] = get_bytes (infp, 1);
00348         color_map [i][1] = get_bytes (infp, 1);
00349         color_map [i][2] = get_bytes (infp, 1);
00350         color_map [i][3] = get_bytes (infp, 1);
00351     }
00352     /* Skip remaining color table entries if more than 2 */
00353     while (i < true_colors) {
00354         (void) get_bytes (infp, 4);
00355         i++;
00356     }
00357
00358     if (color_map [0][0] >= 128) image_xor = 0xFF; /* Invert colors */
00359 }
00360
00361 if (verbose) {
00362     fprintf(stderr, "Color Palette [R, G, B, %s] Values:\n",
00363         (dib_length <= 40) ? "reserved" : "Alpha");
00364     for (i = 0; i < 2; i++) {
00365         fprintf(stderr, "%7d: [", i);
00366         fprintf(stderr, "%3d,", color_map [i][0] & 0xFF);
00367         fprintf(stderr, "%3d,", color_map [i][1] & 0xFF);
00368         fprintf(stderr, "%3d,", color_map [i][2] & 0xFF);
00369         fprintf(stderr, "%3d]\n", color_map [i][3] & 0xFF);
00370     }
00371     if (image_xor == 0xFF) fprintf(stderr, "Will Invert Colors.\n");
00372     fputc ('\n', stderr);
00373 } /* if (verbose) */
00374
00375 /*
00376 Check format before writing output file.
00377 */
00378 if (image_width != 560 && image_width != 576) {
00379     fprintf(stderr, "\\nUnsupported image width: %d\n", image_width);

```

```

00382     fprintf(stderr, "Width should be 560 or 576 pixels.\n\n");
00383     exit (EXIT_FAILURE);
00384 }
00385
00386 if (image_height != 544) {
00387     fprintf(stderr, "\nUnsupported image height: %d\n", image_height);
00388     fprintf(stderr, "Height should be 544 pixels.\n\n");
00389     exit (EXIT_FAILURE);
00390 }
00391
00392 if (num_planes != 1) {
00393     fprintf(stderr, "\nUnsupported number of planes: %d\n", num_planes);
00394     fprintf(stderr, "Number of planes should be 1.\n\n");
00395     exit (EXIT_FAILURE);
00396 }
00397
00398 if (bits_per_pixel != 1) {
00399     fprintf(stderr, "\nUnsupported number of bits per pixel: %d\n",
00400             bits_per_pixel);
00401     fprintf(stderr, "Bits per pixel should be 1.\n\n");
00402     exit (EXIT_FAILURE);
00403 }
00404
00405 if (compression_method != 0 && compression_method != 11) {
00406     fprintf(stderr, "\nUnsupported compression method: %d\n",
00407             compression_method);
00408     fprintf(stderr, "Compression method should be 1 or 11.\n\n");
00409     exit (EXIT_FAILURE);
00410 }
00411
00412 if (true_colors != 2) {
00413     fprintf(stderr, "\nUnsupported number of colors: %d\n", true_colors);
00414     fprintf(stderr, "Number of colors should be 2.\n\n");
00415     exit (EXIT_FAILURE);
00416 }
00417
00418
00419 /*
00420  If we made it this far, things look okay, so write out
00421  the standard header for image conversion.
00422  */
00423 for (i = 0; i < 62; i++) fputc (standard_header[i], outfp);
00424
00425
00426 /*
00427  Image Data.  Each row must be a multiple of 4 bytes, with
00428  padding at the end of each row if necessary.
00429  */
00430 k = 0; /* byte number within the binary image */
00431 for (i = 0; i < 544; i++) {
00432     /*
00433      If original image is 560 pixels wide (not 576), add
00434      2 white bytes at beginning of row.
00435      */
00436     if (image_width == 560) { /* Insert 2 white bytes */
00437         image_bytes[k++] = 0xFF;
00438         image_bytes[k++] = 0xFF;
00439     }
00440     for (j = 0; j < 70; j++) { /* Copy next 70 bytes */
00441         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
00442     }
00443     /*
00444      If original image is 560 pixels wide (not 576), skip
00445      2 padding bytes at end of row in file because we inserted
00446      2 white bytes at the beginning of the row.
00447      */
00448     if (image_width == 560) {
00449         (void) get_bytes (infp, 2);
00450     }
00451     else { /* otherwise, next 2 bytes are part of the image so copy them */
00452         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
00453         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
00454     }
00455 }
00456
00457
00458 /*
00459  Change the image to match the unihex2bmp.c format if original wasn't
00460  */
00461 if (image_width == 560) {
00462     regrid (image_bytes);

```



```

00463 }
00464
00465 for (i = 0; i < 544 * 576 / 8; i++) {
00466     fputc (image_bytes[i], outfp);
00467 }
00468
00469
00470 /*
00471     Wrap up.
00472 */
00473 fclose (infp);
00474 fclose (outfp);
00475
00476 exit (EXIT_SUCCESS);
00477 }

```

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, 0xff,
00529         /* 8 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00530         /* 16 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00531         /* 24 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00532         /* 32 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00533         /* 40 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00534         /* 48 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00535         /* 56 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00536         /* 64 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00
00537     };
00538
00539
00540     /*
00541         First move "U+" left and insert "00" after it.
00542     */
00543     j = 15; /* rows are written bottom to top, so we'll decrement j */
00544     for (i = 543 - 8; i > 544 - 24; i--) {
00545         offset = 72 * i;
00546         image_bytes[offset + 0] = image_bytes[offset + 2];
00547         image_bytes[offset + 1] = image_bytes[offset + 3];
00548         image_bytes[offset + 2] = image_bytes[offset + 4];

```

```

00549     image_bytes[offset + 3] = image_bytes[offset + 4] =
00550     ~zero_pattern[15 - j--] & 0xFF;
00551 }
00552
00553 /*
00554  * Now move glyph bitmaps to the right by 8 pixels.
00555  */
00556 for (i = 0; i < 16; i++) { /* for each glyph row */
00557     for (j = 0; j < 16; j++) { /* for each glyph column */
00558         /* set offset to lower left-hand byte of next glyph */
00559         offset = (32 * 72 * i) + (9 * 72) + (4 * j) + 8;
00560         for (k = 0; k < 16; k++) { /* for each glyph row */
00561             glyph_row = (image_bytes[offset + 0] << 24) |
00562                 (image_bytes[offset + 1] << 16) |
00563                 (image_bytes[offset + 2] << 8) |
00564                 (image_bytes[offset + 3]);
00565             last_pixel = glyph_row & 1; /* preserve border */
00566             glyph_row >>= 4;
00567             glyph_row &= 0xFFFFFEE;
00568             /* Set left 4 pixels to white and preserve last pixel */
00569             glyph_row |= 0xF000000 | last_pixel;
00570             image_bytes[offset + 3] = glyph_row & 0xFF;
00571             glyph_row >>= 8;
00572             image_bytes[offset + 2] = glyph_row & 0xFF;
00573             glyph_row >>= 8;
00574             image_bytes[offset + 1] = glyph_row & 0xFF;
00575             glyph_row >>= 8;
00576             image_bytes[offset + 0] = glyph_row & 0xFF;
00577             offset += 72; /* move up to next row in current glyph */
00578         }
00579     }
00580 }
00581
00582 /* Replace horizontal grid with unihex2bmp.c grid */
00583 for (i = 0; i <= 16; i++) {
00584     offset = 32 * 72 * i;
00585     for (j = 0; j < 72; j++) {
00586         image_bytes[offset + j] = hgrid[j];
00587     }
00588 }
00589
00590 return;
00591 }

```

Here is the caller graph for this function:

5.14 unibmpbump.c

[Go to the documentation of this file.](#)

```

00001 /**
00002  * @file unibmpbump.c
00003  *
00004  * @brief unibmpbump - Adjust a Microsoft bitmap (.bmp) file that
00005  *         was created by unihex2png but converted to .bmp
00006  *
00007  * @author Paul Hardy, unifoundry <at> unifoundry.com
00008  *
00009  * @copyright Copyright (C) 2019 Paul Hardy
00010  *
00011  * This program shifts the glyphs in a bitmap file to adjust an
00012  * original PNG file that was saved in BMP format. This is so the
00013  * result matches the format of a unihex2bmp image. This conversion
00014  * then lets unibmp2hex decode the result.
00015  *
00016  * Synopsis: unibmpbump [-iin_file.bmp] [-oout_file.bmp]
00017  */
00018 /*
00019  * LICENSE:
00020  *
00021  * This program is free software: you can redistribute it and/or modify
00022  * it under the terms of the GNU General Public License as published by
00023  * the Free Software Foundation, either version 2 of the License, or
00024  * (at your option) any later version.
00025  */

```

```

00026     This program is distributed in the hope that it will be useful,
00027     but WITHOUT ANY WARRANTY; without even the implied warranty of
00028     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00029     GNU General Public License for more details.
00030
00031     You should have received a copy of the GNU General Public License
00032     along with this program. If not, see <http://www.gnu.org/licenses/>.
00033 */
00034 #include <stdio.h>
00035 #include <string.h>
00036 #include <stdlib.h>
00037
00038 #define VERSION "1.0"    ///< Version of this program
00039
00040 #define MAX_COMPRESSION_METHOD 13    ///< Maximum supported compression method
00041
00042
00043 /**
00044  @brief The main function.
00045
00046  @param[in] argc The count of command line arguments.
00047  @param[in] argv Pointer to array of command line arguments.
00048  @return This program exits with status EXIT_SUCCESS.
00049 */
00050 int main (int argc, char *argv[]) {
00051
00052     /*
00053      Values preserved from file header (first 14 bytes).
00054     */
00055     char file_format[3];    /* "BM" for original Windows format */
00056     unsigned filesize;    /* size of file in bytes */
00057     unsigned char rsvd_hdr[4];    /* 4 reserved bytes */
00058     unsigned image_start;    /* byte offset of image in file */
00059
00060     /*
00061      Values preserved from Device Independent Bitmap (DIB) Header.
00062
00063      The DIB fields below are in the standard 40-byte header. Version
00064      4 and version 5 headers have more information, mainly for color
00065      information. That is skipped over, because a valid glyph image
00066      is just monochrome.
00067     */
00068     int dib_length;    /* in bytes, for parsing by header version */
00069     int image_width = 0;    /* Signed image width */
00070     int image_height = 0;    /* Signed image height */
00071     int num_planes = 0;    /* number of planes; must be 1 */
00072     int bits_per_pixel = 0;    /* for palletized color maps (< 2^16 colors) */
00073
00074     /*
00075      The following fields are not in the original spec, so initialize
00076      them to 0 so we can correctly parse an original file format.
00077     */
00077     int compression_method=0;    /* 0 --> uncompressed RGB/monochrome */
00078     int image_size = 0;    /* 0 is a valid size if no compression */
00079     int hres = 0;    /* image horizontal resolution */
00080     int vres = 0;    /* image vertical resolution */
00081     int num_colors = 0;    /* Number of colors for palletized images */
00082     int important_colors = 0;    /* Number of significant colors (0 or 2) */
00083
00084     int true_colors = 0;    /* interpret num_colors, which can equal 0 */
00085
00086     /*
00087      Color map. This should be a monochrome file, so only two
00088      colors are stored.
00089     */
00090     unsigned char color_map[2][4];    /* two of R, G, B, and possibly alpha */
00091
00092     /*
00093      The monochrome image bitmap, stored as a vector 544 rows by
00094      72*8 columns.
00095     */
00096     unsigned image_bytes[544*72];
00097
00098     /*
00099      Flags for conversion & I/O.
00100     */
00101     int verbose = 0;    /* Whether to print file info on stderr */
00102     unsigned image_xor = 0x00;    /* Invert (= 0xFF) if color 0 is not black */
00103
00104     /*
00105      Temporary variables.
00106     */

```

```

00107 int i, j, k;          /* loop variables */
00108
00109 /* Compression type, for parsing file */
00110 char *compression_type[MAX_COMPRESSION_METHOD + 1] = {
00111     "BI_RGB",          /* 0 */
00112     "BI_RLE8",         /* 1 */
00113     "BI_RLE4",         /* 2 */
00114     "BI_BITFIELDS",    /* 3 */
00115     "BI_JPEG",         /* 4 */
00116     "BI_PNG",          /* 5 */
00117     "BI_ALPHABITFIELDS", /* 6 */
00118     "", "", "", "",    /* 7 - 10 */
00119     "BI_CMYK",         /* 11 */
00120     "BI_CMYKRLE8",     /* 12 */
00121     "BI_CMYKRLE4",     /* 13 */
00122 };
00123
00124 /* Standard unihex2bmp.c header for BMP image */
00125 unsigned standard_header [62] = {
00126     /* 0 */ 0x42, 0x4d, 0x3e, 0x99, 0x00, 0x00, 0x00, 0x00,
00127     /* 8 */ 0x00, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x28, 0x00,
00128     /* 16 */ 0x00, 0x00, 0x40, 0x02, 0x00, 0x00, 0x20, 0x02,
00129     /* 24 */ 0x00, 0x00, 0x01, 0x00, 0x01, 0x00, 0x00, 0x00,
00130     /* 32 */ 0x00, 0x00, 0x00, 0x99, 0x00, 0x00, 0xc4, 0x0e,
00131     /* 40 */ 0x00, 0x00, 0xc4, 0x0e, 0x00, 0x00, 0x00, 0x00,
00132     /* 48 */ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
00133     /* 56 */ 0x00, 0x00, 0xff, 0xff, 0xff, 0x00,
00134 };
00135
00136 unsigned get_bytes (FILE *, int);
00137 void regrid (unsigned *);
00138
00139 char *infile="", *outfile=""; /* names of input and output files */
00140 FILE *infp, *outfp;          /* file pointers of input and output files */
00141
00142 /*
00143  Process command line arguments.
00144 */
00145 if (argc > 1) {
00146     for (i = 1; i < argc; i++) {
00147         if (argv[i][0] == '-') { /* this is an option argument */
00148             switch (argv[i][1]) {
00149                 case 'i': /* name of input file */
00150                     infile = &argv[i][2];
00151                     break;
00152                 case 'o': /* name of output file */
00153                     outfile = &argv[i][2];
00154                     break;
00155                 case 'v': /* verbose output */
00156                     verbose = 1;
00157                     break;
00158                 case 'V': /* print version & quit */
00159                     fprintf (stderr, "unibmpbump version %s\n\n", VERSION);
00160                     exit (EXIT_SUCCESS);
00161                     break;
00162                 case '-': /* see if "--verbose" */
00163                     if (strcmp (argv[i], "--verbose") == 0) {
00164                         verbose = 1;
00165                     }
00166                     else if (strcmp (argv[i], "--version") == 0) {
00167                         fprintf (stderr, "unibmpbump version %s\n\n", VERSION);
00168                         exit (EXIT_SUCCESS);
00169                     }
00170                     break;
00171                 default: /* if unrecognized option, print list and exit */
00172                     fprintf (stderr, "\nSyntax:\n\n");
00173                     fprintf (stderr, "    unibmpbump");
00174                     fprintf (stderr, "-i<Input_File> -o<Output_File>\n\n");
00175                     fprintf (stderr, "-v or --verbose gives verbose output");
00176                     fprintf (stderr, " on stderr\n\n");
00177                     fprintf (stderr, "-V or --version prints version");
00178                     fprintf (stderr, " on stderr and exits\n\n");
00179                     fprintf (stderr, "\nExample:\n\n");
00180                     fprintf (stderr, "    unibmpbump -iuni0101.bmp");
00181                     fprintf (stderr, " -onew-uni0101.bmp\n\n");
00182                     exit (EXIT_SUCCESS);
00183             }
00184         }
00185     }
00186 }
00187

```

```

00188  /*
00189      Make sure we can open any I/O files that were specified before
00190      doing anything else.
00191  */
00192  if (strlen (infile) > 0) {
00193      if ((infp = fopen (infile, "r")) == NULL) {
00194          fprintf (stderr, "Error: can't open %s for input.\n", infile);
00195          exit (EXIT_FAILURE);
00196      }
00197  }
00198  else {
00199      infp = stdin;
00200  }
00201  if (strlen (outfile) > 0) {
00202      if ((outfp = fopen (outfile, "w")) == NULL) {
00203          fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00204          exit (EXIT_FAILURE);
00205      }
00206  }
00207  else {
00208      outfp = stdout;
00209  }
00210
00211
00212  /* Read bitmap file header */
00213  file_format[0] = get_bytes (infp, 1);
00214  file_format[1] = get_bytes (infp, 1);
00215  file_format[2] = '\0'; /* Terminate string with null */
00216
00217  /* Read file size */
00218  filesize = get_bytes (infp, 4);
00219
00220  /* Read Reserved bytes */
00221  rsvd_hdr[0] = get_bytes (infp, 1);
00222  rsvd_hdr[1] = get_bytes (infp, 1);
00223  rsvd_hdr[2] = get_bytes (infp, 1);
00224  rsvd_hdr[3] = get_bytes (infp, 1);
00225
00226  /* Read Image Offset Address within file */
00227  image_start = get_bytes (infp, 4);
00228
00229  /*
00230      See if this looks like a valid image file based on
00231      the file header first two bytes.
00232  */
00233  if (strncmp (file_format, "BM", 2) != 0) {
00234      fprintf (stderr, "\nInvalid file format: not file type \"BM\".\n\n");
00235      exit (EXIT_FAILURE);
00236  }
00237
00238  if (verbose) {
00239      fprintf (stderr, "\nFile Header:\n");
00240      fprintf (stderr, "  File Type:  \"%s\"\n", file_format);
00241      fprintf (stderr, "  File Size:  %d bytes\n", filesize);
00242      fprintf (stderr, "  Reserved:  ");
00243      for (i = 0; i < 4; i++) fprintf (stderr, " 0x%02X", rsvd_hdr[i]);
00244      fputc ('\n', stderr);
00245      fprintf (stderr, "  Image Start: %d. = 0x%02X = 0%05o\n\n",
00246              image_start, image_start, image_start);
00247  } /* if (verbose) */
00248
00249  /*
00250      Device Independent Bitmap (DIB) Header: bitmap information header
00251      ("BM" format file DIB Header is 12 bytes long).
00252  */
00253  dib_length = get_bytes (infp, 4);
00254
00255  /*
00256      Parse one of three versions of Device Independent Bitmap (DIB) format:
00257
00258          Length  Format
00259          -----
00260             12  BITMAPCOREHEADER
00261             40  BITMAPINFOHEADER
00262             108 BITMAPV4HEADER
00263             124 BITMAPV5HEADER
00264  */
00265  if (dib_length == 12) { /* BITMAPCOREHEADER format -- UNTESTED */
00266      image_width  = get_bytes (infp, 2);
00267      image_height = get_bytes (infp, 2);
00268      num_planes   = get_bytes (infp, 2);

```

```

00269     bits_per_pixel = get_bytes (infp, 2);
00270 }
00271 else if (dib_length >= 40) { /* BITMAPINFOHEADER format or later */
00272     image_width = get_bytes (infp, 4);
00273     image_height = get_bytes (infp, 4);
00274     num_planes = get_bytes (infp, 2);
00275     bits_per_pixel = get_bytes (infp, 2);
00276     compression_method = get_bytes (infp, 4); /* BI_BITFIELDS */
00277     image_size = get_bytes (infp, 4);
00278     hres = get_bytes (infp, 4);
00279     vres = get_bytes (infp, 4);
00280     num_colors = get_bytes (infp, 4);
00281     important_colors = get_bytes (infp, 4);
00282
00283     /* true_colors is true number of colors in image */
00284     if (num_colors == 0)
00285         true_colors = 1 « bits_per_pixel;
00286     else
00287         true_colors = num_colors;
00288
00289     /*
00290     If dib_length > 40, the format is BITMAPV4HEADER or
00291     BITMAPV5HEADER. As this program is only designed
00292     to handle a monochrome image, we can ignore the rest
00293     of the header but must read past the remaining bytes.
00294     */
00295     for (i = 40; i < dib_length; i++) (void)get_bytes (infp, 1);
00296 }
00297
00298 if (verbose) {
00299     fprintf (stderr, "Device Independent Bitmap (DIB) Header:\n");
00300     fprintf (stderr, "  DIB Length: %9d bytes (version = ", dib_length);
00301
00302     if (dib_length == 12) fprintf (stderr, "\"BITMAPCOREHEADER\");\n");
00303     else if (dib_length == 40) fprintf (stderr, "\"BITMAPINFOHEADER\");\n");
00304     else if (dib_length == 108) fprintf (stderr, "\"BITMAPV4HEADER\");\n");
00305     else if (dib_length == 124) fprintf (stderr, "\"BITMAPV5HEADER\");\n");
00306     else fprintf (stderr, "unknown");
00307     fprintf (stderr, "  Bitmap Width: %6d pixels\n", image_width);
00308     fprintf (stderr, "  Bitmap Height: %6d pixels\n", image_height);
00309     fprintf (stderr, "  Color Planes: %6d\n", num_planes);
00310     fprintf (stderr, "  Bits per Pixel: %6d\n", bits_per_pixel);
00311     fprintf (stderr, "  Compression Method: %2d --> ", compression_method);
00312     if (compression_method <= MAX_COMPRESSION_METHOD) {
00313         fprintf (stderr, "%s", compression_type [compression_method]);
00314     }
00315     /*
00316     Supported compression method values:
00317         0 --> uncompressed RGB
00318         11 --> uncompressed CMYK
00319     */
00320     if (compression_method == 0 || compression_method == 11) {
00321         fprintf (stderr, " (no compression)");
00322     }
00323     else {
00324         fprintf (stderr, "Image uses compression; this is unsupported.\n\n");
00325         exit (EXIT_FAILURE);
00326     }
00327     fprintf (stderr, "\n");
00328     fprintf (stderr, "  Image Size: %5d bytes\n", image_size);
00329     fprintf (stderr, "  Horizontal Resolution: %5d pixels/meter\n", hres);
00330     fprintf (stderr, "  Vertical Resolution: %5d pixels/meter\n", vres);
00331     fprintf (stderr, "  Number of Colors: %5d", num_colors);
00332     if (num_colors != true_colors) {
00333         fprintf (stderr, " --> %d", true_colors);
00334     }
00335     fputc ('\n', stderr);
00336     fprintf (stderr, "  Important Colors: %5d", important_colors);
00337     if (important_colors == 0)
00338         fprintf (stderr, " (all colors are important)");
00339     fprintf (stderr, "\n\n");
00340 } /* if (verbose) */
00341
00342 /*
00343 Print Color Table information for images with pallettized colors.
00344 */
00345 if (bits_per_pixel <= 8) {
00346     for (i = 0; i < 2; i++) {
00347         color_map [i][0] = get_bytes (infp, 1);
00348         color_map [i][1] = get_bytes (infp, 1);
00349         color_map [i][2] = get_bytes (infp, 1);

```

```

00350     color_map [i][3] = get_bytes (infp, 1);
00351 }
00352 /* Skip remaining color table entries if more than 2 */
00353 while (i < true_colors) {
00354     (void) get_bytes (infp, 4);
00355     i++;
00356 }
00357
00358 if (color_map [0][0] >= 128) image_xor = 0xFF; /* Invert colors */
00359 }
00360
00361 if (verbose) {
00362     fprintf (stderr, "Color Palette [R, G, B, %s] Values:\n",
00363             (dib_length <= 40) ? "reserved" : "Alpha");
00364     for (i = 0; i < 2; i++) {
00365         fprintf (stderr, "%7d: [", i);
00366         fprintf (stderr, "%3d,", color_map [i][0] & 0xFF);
00367         fprintf (stderr, "%3d,", color_map [i][1] & 0xFF);
00368         fprintf (stderr, "%3d,", color_map [i][2] & 0xFF);
00369         fprintf (stderr, "%3d]\n", color_map [i][3] & 0xFF);
00370     }
00371     if (image_xor == 0xFF) fprintf (stderr, "Will Invert Colors.\n");
00372     fputc ('\n', stderr);
00373 }
00374 /* if (verbose) */
00375
00376 /*
00377  * Check format before writing output file.
00378  */
00379 if (image_width != 560 && image_width != 576) {
00380     fprintf (stderr, "\nUnsupported image width: %d\n", image_width);
00381     fprintf (stderr, "Width should be 560 or 576 pixels.\n\n");
00382     exit (EXIT_FAILURE);
00383 }
00384
00385 if (image_height != 544) {
00386     fprintf (stderr, "\nUnsupported image height: %d\n", image_height);
00387     fprintf (stderr, "Height should be 544 pixels.\n\n");
00388     exit (EXIT_FAILURE);
00389 }
00390
00391 if (num_planes != 1) {
00392     fprintf (stderr, "\nUnsupported number of planes: %d\n", num_planes);
00393     fprintf (stderr, "Number of planes should be 1.\n\n");
00394     exit (EXIT_FAILURE);
00395 }
00396
00397 if (bits_per_pixel != 1) {
00398     fprintf (stderr, "\nUnsupported number of bits per pixel: %d\n",
00399             bits_per_pixel);
00400     fprintf (stderr, "Bits per pixel should be 1.\n\n");
00401     exit (EXIT_FAILURE);
00402 }
00403
00404 if (compression_method != 0 && compression_method != 11) {
00405     fprintf (stderr, "\nUnsupported compression method: %d\n",
00406             compression_method);
00407     fprintf (stderr, "Compression method should be 1 or 11.\n\n");
00408     exit (EXIT_FAILURE);
00409 }
00410
00411 if (true_colors != 2) {
00412     fprintf (stderr, "\nUnsupported number of colors: %d\n", true_colors);
00413     fprintf (stderr, "Number of colors should be 2.\n\n");
00414     exit (EXIT_FAILURE);
00415 }
00416
00417 /*
00418  * If we made it this far, things look okay, so write out
00419  * the standard header for image conversion.
00420  */
00421 for (i = 0; i < 62; i++) fputc (standard_header[i], outfp);
00422
00423 /*
00424  * Image Data. Each row must be a multiple of 4 bytes, with
00425  * padding at the end of each row if necessary.
00426  */
00427 k = 0; /* byte number within the binary image */

```

```

00431 for (i = 0; i < 544; i++) {
00432     /*
00433      * If original image is 560 pixels wide (not 576), add
00434      * 2 white bytes at beginning of row.
00435      */
00436     if (image_width == 560) { /* Insert 2 white bytes */
00437         image_bytes[k++] = 0xFF;
00438         image_bytes[k++] = 0xFF;
00439     }
00440     for (j = 0; j < 70; j++) { /* Copy next 70 bytes */
00441         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
00442     }
00443     /*
00444      * If original image is 560 pixels wide (not 576), skip
00445      * 2 padding bytes at end of row in file because we inserted
00446      * 2 white bytes at the beginning of the row.
00447      */
00448     if (image_width == 560) {
00449         (void) get_bytes (infp, 2);
00450     }
00451     else { /* otherwise, next 2 bytes are part of the image so copy them */
00452         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
00453         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
00454     }
00455 }
00456
00457
00458 /*
00459  * Change the image to match the unihex2bmp.c format if original wasn't
00460  */
00461 if (image_width == 560) {
00462     regrid (image_bytes);
00463 }
00464
00465 for (i = 0; i < 544 * 576 / 8; i++) {
00466     fputc (image_bytes[i], outfp);
00467 }
00468
00469
00470 /*
00471  * Wrap up.
00472  */
00473 fclose (infp);
00474 fclose (outfp);
00475
00476 exit (EXIT_SUCCESS);
00477 }
00478
00479
00480 /**
00481  * @brief Get from 1 to 4 bytes, inclusive, from input file.
00482
00483  * @param[in] infp Pointer to input file.
00484  * @param[in] nbytes Number of bytes to read, from 1 to 4, inclusive.
00485  * @return The unsigned 1 to 4 bytes in machine native endian format.
00486  */
00487 unsigned get_bytes (FILE *infp, int nbytes) {
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 }
00504
00505
00506 /**
00507  * @brief After reading in the image, shift it.
00508
00509  * This function adjusts the input image from an original PNG file
00510  * to match unihex2bmp.c format.
00511

```



```

00512  @param[in,out] image_bytes The pixels in an image.
00513 */
00514 void regrid (unsigned *image_bytes) {
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, 0xff,
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      * First move "U+" left and insert "00" after it.
00541      */
00542     j = 15; /* rows are written bottom to top, so we'll decrement j */
00543     for (i = 543 - 8; i > 544 - 24; i--) {
00544         offset = 72 * i;
00545         image_bytes[offset + 0] = image_bytes[offset + 2];
00546         image_bytes[offset + 1] = image_bytes[offset + 3];
00547         image_bytes[offset + 2] = image_bytes[offset + 4];
00548         image_bytes[offset + 3] = image_bytes[offset + 4] =
00549             ~zero_pattern[15 - j--] & 0xFF;
00550     }
00551
00552     /*
00553      * Now move glyph bitmaps to the right by 8 pixels.
00554      */
00555     for (i = 0; i < 16; i++) { /* for each glyph row */
00556         for (j = 0; j < 16; j++) { /* for each glyph column */
00557             /* set offset to lower left-hand byte of next glyph */
00558             offset = (32 * 72 * i) + (9 * 72) + (4 * j) + 8;
00559             for (k = 0; k < 16; k++) { /* for each glyph row */
00560                 glyph_row = (image_bytes[offset + 0] << 24) |
00561                     (image_bytes[offset + 1] << 16) |
00562                     (image_bytes[offset + 2] << 8) |
00563                     (image_bytes[offset + 3]);
00564                 last_pixel = glyph_row & 1; /* preserve border */
00565                 glyph_row >>= 4;
00566                 glyph_row &= 0x0FFFFFFF;
00567                 /* Set left 4 pixels to white and preserve last pixel */
00568                 glyph_row |= 0xF0000000 | last_pixel;
00569                 image_bytes[offset + 3] = glyph_row & 0xFF;
00570                 glyph_row >>= 8;
00571                 image_bytes[offset + 2] = glyph_row & 0xFF;
00572                 glyph_row >>= 8;
00573                 image_bytes[offset + 1] = glyph_row & 0xFF;
00574                 glyph_row >>= 8;
00575                 image_bytes[offset + 0] = glyph_row & 0xFF;
00576                 offset += 72; /* move up to next row in current glyph */
00577             }
00578         }
00579     }
00580
00581     /* Replace horizontal grid with unihex2bmp.c grid */
00582     for (i = 0; i <= 16; i++) {
00583         offset = 32 * 72 * i;
00584         for (j = 0; j < 72; j++) {
00585             image_bytes[offset + j] = hgrid[j];
00586         }
00587     }
00588
00589     return;
00590 }
00591 }

```

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, unsigned *cstart, unsigned *cend, char *coverstring)`
Get next Unicode range.
- `void print_subtotal (FILE *outfp, int print_n, int nglyphs, unsigned cstart, unsigned 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 63 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 74 of file [unicoverage.c](#).

```
00075 {
00076
00077     int    print_n=0;        /* print # of glyphs, not percentage */
00078     unsigned i;              /* loop variable */
00079     unsigned slen;           /* string length of coverage file line */
00080     char    inbuf[256];      /* input buffer */
00081     unsigned thischar;       /* the current character */
00082
00083     char *infile="", *outfile=""; /* names of input and output files */
00084     FILE *infp, *outfp;      /* file pointers of input and output files */
00085     FILE *coveragefp;        /* file pointer to coverage.dat file */
00086     unsigned cstart, cend;    /* current coverage start and end code points */
00087     char coverstring[MAXBUF]; /* description of current coverage range */
00088     int nglyphs;              /* number of glyphs in this section */
00089
00090     /* to get next range & name of Unicode glyphs */
00091     int nextrange (FILE *coveragefp, unsigned *cstart, unsigned *cend,
00092                  char *coverstring);
00093
00094     void print_subtotal (FILE *outfp, int print_n, int nglyphs,
00095                        unsigned cstart, unsigned cend, char *coverstring);
00096
00097     if ((coveragefp = fopen ("coverage.dat", "r")) == NULL) {
00098         fprintf (stderr, "\nError: data file \"coverage.dat\" not found.\n\n");
00099         exit (0);
00100     }
```

```

00101
00102 if (argc > 1) {
00103     for (i = 1; i < argc; i++) {
00104         if (argv[i][0] == '-') { /* this is an option argument */
00105             switch (argv[i][1]) {
00106                 case 'i': /* name of input file */
00107                     infile = &argv[i][2];
00108                     break;
00109                 case 'n': /* print number of glyphs instead of percentage */
00110                     print_n = 1;
00111                 case 'o': /* name of output file */
00112                     outfile = &argv[i][2];
00113                     break;
00114                 default: /* if unrecognized option, print list and exit */
00115                     fprintf (stderr, "\nSyntax:\n\n");
00116                     fprintf (stderr, " %s -p<Unicode_Page> ", argv[0]);
00117                     fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
00118                     exit (1);
00119             }
00120         }
00121     }
00122 }
00123 /*
00124    Make sure we can open any I/O files that were specified before
00125    doing anything else.
00126 */
00127 if (strlen (infile) > 0) {
00128     if ((infp = fopen (infile, "r")) == NULL) {
00129         fprintf (stderr, "Error: can't open %s for input.\n", infile);
00130         exit (1);
00131     }
00132 }
00133 else {
00134     infp = stdin;
00135 }
00136 if (strlen (outfile) > 0) {
00137     if ((outfp = fopen (outfile, "w")) == NULL) {
00138         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00139         exit (1);
00140     }
00141 }
00142 else {
00143     outfp = stdout;
00144 }
00145
00146 /*
00147    Print header row.
00148 */
00149 if (print_n) {
00150     fprintf (outfp, "# Glyphs      Range      Script\n");
00151     fprintf (outfp, "-----      ----      ----\n");
00152 }
00153 else {
00154     fprintf (outfp, "Covered      Range      Script\n");
00155     fprintf (outfp, "-----      ----      ----\n");
00156 }
00157
00158 slen = nextrange (coveragefp, &cstart, &cend, coverstring);
00159 nglyphs = 0;
00160
00161 /*
00162    Read in the glyphs in the file
00163 */
00164 while (slen != 0 && fgets (inbuf, MAXBUF-1, infp) != NULL) {
00165     sscanf (inbuf, "%x", &thischar);
00166
00167     /* Read a character beyond end of current script. */
00168     while (cend < thischar && slen != 0) {
00169         print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);
00170
00171         /* start new range total */
00172         slen = nextrange (coveragefp, &cstart, &cend, coverstring);
00173         nglyphs = 0;
00174     }
00175     nglyphs++;
00176 }
00177
00178 print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);
00179
00180 exit (0);
00181 }

```

Here is the call graph for this function:

5.15.3.2 nextrange()

```
int nextrange (
    FILE * coveragefp,
    unsigned * cstart,
    unsigned * 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 196 of file [unicoverage.c](#).

```
00199 {
00200     int i;
00201     static char inbuf[MAXBUF];
00202     int retval;          /* the return value */
00203
00204     retval = 0;
00205
00206     do {
00207         if (fgets (inbuf, MAXBUF-1, coveragefp) != NULL) {
00208             retval = strlen (inbuf);
00209             if ((inbuf[0] >= '0' && inbuf[0] <= '9') ||
00210                 (inbuf[0] >= 'A' && inbuf[0] <= 'F') ||
00211                 (inbuf[0] >= 'a' && inbuf[0] <= 'f')) {
00212                 sscanf (inbuf, "%x-%x", cstart, cend);
00213                 i = 0;
00214                 while (inbuf[i] != ' ') i++; /* find first blank */
00215                 while (inbuf[i] == ' ') i++; /* find next non-blank */
00216                 strncpy (coverstring, &inbuf[i], MAXBUF);
00217             }
00218             else retval = 0;
00219         }
00220         else retval = 0;
00221     } while (retval == 0 && !feof (coveragefp));
00222
00223     return (retval);
00224 }
```

Here is the caller graph for this function:

5.15.3.3 print_subtotal()

```
void print_subtotal (
    FILE * outfp,
    int print_n,
    int nglyphs,
    unsigned cstart,
    unsigned 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 237 of file [unicoverage.c](#).

```
00238                                     {
00239
00240     /* print old range total */
00241     if (print_n) { /* Print number of glyphs, not percentage */
00242         fprintf (outfp, " %6d ", nglyphs);
00243     }
00244     else {
00245         fprintf (outfp, " %5.1f%%", 100.0*nglyphs/(1+cend-cstart));
00246     }
00247
00248     if (cend < 0x10000)
00249         fprintf (outfp, " U+%04X..U+%04X  %s",
00250             cstart, cend, coverstring);
00251     else
00252         fprintf (outfp, " U+%05X..U+%05X  %s",
00253             cstart, cend, coverstring);
00254
00255     return;
00256 }
```

Here is the caller graph for this function:

5.16 unicoverage.c

[Go to the documentation of this file.](#)

```
00001 /**
00002  @file unicoverage.c
00003
00004  @brief unicoverage - Show the coverage of Unicode plane scripts
00005         for a GNU Unifont hex glyph file
00006
00007  @author Paul Hardy, unifoundry <at> unifoundry.com, 6 January 2008
00008
00009  @copyright Copyright (C) 2008, 2013 Paul Hardy
00010
00011  Synopsis: unicoverage [-ifont_file.hex] [-ocoverage_file.txt]
00012
```

```

00013  This program requires the file "coverage.dat" to be present
00014  in the directory from which it is run.
00015  */
00016  /*
00017  LICENSE:
00018
00019  This program is free software: you can redistribute it and/or modify
00020  it under the terms of the GNU General Public License as published by
00021  the Free Software Foundation, either version 2 of the License, or
00022  (at your option) any later version.
00023
00024  This program is distributed in the hope that it will be useful,
00025  but WITHOUT ANY WARRANTY; without even the implied warranty of
00026  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00027  GNU General Public License for more details.
00028
00029  You should have received a copy of the GNU General Public License
00030  along with this program. If not, see <http://www.gnu.org/licenses/>.
00031  */
00032  /*
00033  /*
00034  2016 (Paul Hardy): Modified in Unifont 9.0.01 release to remove non-existent
00035  "-p" option and empty example from help printout.
00036
00037  2018 (Paul Hardy): Modified to cover entire Unicode range, not just Plane 0.
00038
00039  11 May 2019: [Paul Hardy] changed strcpy function call to strncpy
00040  for better error handling.
00041
00042  31 May 2019: [Paul Hardy] replaced strncpy call with strncpy
00043  for compilation on more systems.
00044
00045  4 June 2022: [Paul Hardy] Adjusted column spacing for better alignment
00046  of Unicode Plane 1-15 scripts. Added "-n" option to print number of
00047  glyphs in each range instead of percent coverage.
00048
00049  18 September 2022: [Paul Hardy] in nextrange function, initialize retval.
00050
00051  21 October 2023: [Paul Hardy]
00052  Added full function prototype for nextrange function in main function.
00053
00054  6 September 2025: [Paul Hardy] changed cstart and cend from int to
00055  unsigned int to match sscanf parameter declarations.
00056  */
00057
00058  #include <stdio.h>
00059  #include <stdlib.h>
00060  #include <string.h>
00061
00062
00063  #define MAXBUF 256 ///< Maximum input line length - 1
00064
00065
00066  /**
00067   @brief The main function.
00068
00069   @param[in] argc The count of command line arguments.
00070   @param[in] argv Pointer to array of command line arguments.
00071   @return This program exits with status 0.
00072  */
00073  int
00074  main (int argc, char *argv[])
00075  {
00076
00077      int print_n=0; /* print # of glyphs, not percentage */
00078      unsigned i; /* loop variable */
00079      unsigned slen; /* string length of coverage file line */
00080      char inbuf[256]; /* input buffer */
00081      unsigned thischar; /* the current character */
00082
00083      char *infile="", *outfile=""; /* names of input and output files */
00084      FILE *infp, *outfp; /* file pointers of input and output files */
00085      FILE *coveragefp; /* file pointer to coverage.dat file */
00086      unsigned cstart, cend; /* current coverage start and end code points */
00087      char coverstring[MAXBUF]; /* description of current coverage range */
00088      int nglyphs; /* number of glyphs in this section */
00089
00090      /* to get next range & name of Unicode glyphs */
00091      int nextrange (FILE *coveragefp, unsigned *cstart, unsigned *cend,
00092                    char *coverstring);
00093

```

```

00094 void print_subtotal (FILE *outfp, int print_n, int nglyphs,
00095                     unsigned cstart, unsigned cend, char *coverstring);
00096
00097 if ((coveragefp = fopen ("coverage.dat", "r")) == NULL) {
00098     fprintf (stderr, "\nError: data file \"coverage.dat\" not found.\n\n");
00099     exit (0);
00100 }
00101
00102 if (argc > 1) {
00103     for (i = 1; i < argc; i++) {
00104         if (argv[i][0] == '-') { /* this is an option argument */
00105             switch (argv[i][1]) {
00106                 case 'i': /* name of input file */
00107                     infile = &argv[i][2];
00108                     break;
00109                 case 'n': /* print number of glyphs instead of percentage */
00110                     print_n = 1;
00111                 case 'o': /* name of output file */
00112                     outfile = &argv[i][2];
00113                     break;
00114                 default: /* if unrecognized option, print list and exit */
00115                     fprintf (stderr, "\nSyntax:\n\n");
00116                     fprintf (stderr, " %s -p<Unicode_Page> ", argv[0]);
00117                     fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
00118                     exit (1);
00119             }
00120         }
00121     }
00122 }
00123 /*
00124  Make sure we can open any I/O files that were specified before
00125  doing anything else.
00126 */
00127 if (strlen (infile) > 0) {
00128     if ((infp = fopen (infile, "r")) == NULL) {
00129         fprintf (stderr, "Error: can't open %s for input.\n", infile);
00130         exit (1);
00131     }
00132 }
00133 else {
00134     infp = stdin;
00135 }
00136 if (strlen (outfile) > 0) {
00137     if ((outfp = fopen (outfile, "w")) == NULL) {
00138         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00139         exit (1);
00140     }
00141 }
00142 else {
00143     outfp = stdout;
00144 }
00145
00146 /*
00147  Print header row.
00148 */
00149 if (print_n) {
00150     fprintf (outfp, "# Glyphs      Range      Script\n");
00151     fprintf (outfp, "-----      ----      ----\n");
00152 }
00153 else {
00154     fprintf (outfp, "Covered      Range      Script\n");
00155     fprintf (outfp, "-----      ----      ----\n");
00156 }
00157
00158 slen = nextrange (coveragefp, &cstart, &cend, coverstring);
00159 nglyphs = 0;
00160
00161 /*
00162  Read in the glyphs in the file
00163 */
00164 while (slen != 0 && fgets (inbuf, MAXBUF-1, infp) != NULL) {
00165     sscanf (inbuf, "%x", &thischar);
00166
00167     /* Read a character beyond end of current script. */
00168     while (cend < thischar && slen != 0) {
00169         print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);
00170
00171         /* start new range total */
00172         slen = nextrange (coveragefp, &cstart, &cend, coverstring);
00173         nglyphs = 0;
00174     }

```



```

00175     nglyphs++;
00176 }
00177
00178 print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);
00179
00180 exit (0);
00181 }
00182
00183 /**
00184  @brief Get next Unicode range.
00185
00186  This function reads the next Unicode script range to count its
00187  glyph coverage.
00188
00189  @param[in] coveragefp File pointer to Unicode script range data file.
00190  @param[in] cstart Starting code point in current Unicode script range.
00191  @param[in] cend Ending code point in current Unicode script range.
00192  @param[out] coverstring String containing <cstart>-<cend> substring.
00193  @return Length of the last string read, or 0 for end of file.
00194 */
00195 int
00196 nextrange (FILE *coveragefp,
00197            unsigned *cstart, unsigned *cend,
00198            char *coverstring)
00199 {
00200     int i;
00201     static char inbuf[MAXBUF];
00202     int retval; /* the return value */
00203
00204     retval = 0;
00205
00206     do {
00207         if (fgets (inbuf, MAXBUF-1, coveragefp) != NULL) {
00208             retval = strlen (inbuf);
00209             if ((inbuf[0] >= '0' && inbuf[0] <= '9') ||
00210                 (inbuf[0] >= 'A' && inbuf[0] <= 'F') ||
00211                 (inbuf[0] >= 'a' && inbuf[0] <= 'f')) {
00212                 sscanf (inbuf, "%x-%x", cstart, cend);
00213                 i = 0;
00214                 while (inbuf[i] != ' ') i++; /* find first blank */
00215                 while (inbuf[i] == ' ') i++; /* find next non-blank */
00216                 strncpy (coverstring, &inbuf[i], MAXBUF);
00217             }
00218             else retval = 0;
00219         }
00220         else retval = 0;
00221     } while (retval == 0 && !feof (coveragefp));
00222
00223     return (retval);
00224 }
00225
00226 /**
00227  @brief Print the subtotal for one Unicode script range.
00228
00229  @param[in] outfp Pointer to output file.
00230  @param[in] print_n 1 = print number of glyphs, 0 = print percentage.
00231  @param[in] nglyphs Number of glyphs in current range.
00232  @param[in] cstart Starting code point for current range.
00233  @param[in] cend Ending code point for current range.
00234  @param[in] coverstring Character string of "<cstart>-<cend>".
00235 */
00236 void
00237 print_subtotal (FILE *outfp, int print_n, int nglyphs,
00238                unsigned cstart, unsigned cend, char *coverstring) {
00239
00240     /* print old range total */
00241     if (print_n) { /* Print number of glyphs, not percentage */
00242         fprintf (outfp, " %6d ", nglyphs);
00243     }
00244     else {
00245         fprintf (outfp, " %5.1f%%", 100.0*nglyphs/(1+cend-cstart));
00246     }
00247
00248     if (cend < 0x10000)
00249         fprintf (outfp, " U+%04X..U+%04X %s",
00250                 cstart, cend, coverstring);
00251     else
00252         fprintf (outfp, " U+%05X..U+%05X %s",
00253                 cstart, cend, coverstring);
00254
00255     return;

```

```
00256 }
```

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

Copyright (C) 2007, 2008, 2013 Paul Hardy

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

Synopsis: unidup < unifont_file.hex

[Hopefully there won't be any output!]

Definition in file [unidup.c](#).

5.17.2 Macro Definition Documentation

5.17.2.1 MAXBUF

```
#define MAXBUF 256
```

Maximum input line length - 1.

Definition at line [43](#) 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 [54](#) of file [unidup.c](#).

```
00055 {
00056
00057     int ix, iy; /* two code points to compare for equality */
00058     char inbuf[MAXBUF];
00059     char *infile; /* the input file name */
00060     FILE *infilep; /* file pointer to input file */
00061
00062     if (argc > 1) {
00063         infile = argv[1];
00064         if ((infilep = fopen (infile, "r")) == NULL) {
00065             fprintf (stderr, "\nERROR: Can't open file %s\n\n", infile);
00066             exit (EXIT_FAILURE);
00067         }
00068     }
00069     else {
00070         infilep = stdin;
00071     }
```

```

00072
00073     ix = -1;
00074
00075     while (fgets (inbuf, MAXBUF-1, infilefp) != NULL) {
00076         sscanf (inbuf, "%X", (unsigned *)&iy);
00077         if (ix == iy) fprintf (stderr, "Duplicate code point: %04X\n", ix);
00078         else ix = iy;
00079     }
00080     exit (0);
00081 }

```

5.18 unidup.c

Go to the documentation of this file.

```

00001 /**
00002     @file unidup.c
00003
00004     @brief unidup - Check for duplicate code points in sorted unifont.hex file
00005
00006     @author Paul Hardy, unifoundry <at> unifoundry.com, December 2007
00007
00008     @copyright Copyright (C) 2007, 2008, 2013 Paul Hardy
00009
00010     This program reads a sorted list of glyphs in Unifont .hex format
00011     and prints duplicate code points on stderr if any were detected.
00012
00013     Synopsis: unidup < unifont__file.hex
00014
00015             [Hopefully there won't be any output!]
00016 */
00017 /*
00018     LICENSE:
00019
00020     This program is free software: you can redistribute it and/or modify
00021     it under the terms of the GNU General Public License as published by
00022     the Free Software Foundation, either version 2 of the License, or
00023     (at your option) any later version.
00024
00025     This program is distributed in the hope that it will be useful,
00026     but WITHOUT ANY WARRANTY; without even the implied warranty of
00027     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00028     GNU General Public License for more details.
00029
00030     You should have received a copy of the GNU General Public License
00031     along with this program. If not, see <http://www.gnu.org/licenses/>.
00032 */
00033
00034 /*
00035     6 September 2025 [Paul Hardy]:
00036     - Changed iy from "int" to "unsigned" for compatibility with
00037       sscanf definition.
00038 */
00039
00040 #include <stdio.h>
00041 #include <stdlib.h>
00042
00043 #define MAXBUF 256 ///< Maximum input line length - 1
00044
00045
00046 /**
00047     @brief The main function.
00048
00049     @param[in] argc The count of command line arguments.
00050     @param[in] argv Pointer to array of command line arguments.
00051     @return This program exits with status 0.
00052 */
00053 int
00054 main (int argc, char **argv)
00055 {
00056
00057     int ix, iy; /* two code points to compare for equality */
00058     char inbuf[MAXBUF];
00059     char *infile; /* the input file name */
00060     FILE *infilefp; /* file pointer to input file */
00061

```

```

00062  if (argc > 1) {
00063      infile = argv[1];
00064      if ((infilep = fopen (infile, "r")) == NULL) {
00065          fprintf (stderr, "\nERROR: Can't open file %s\n", infile);
00066          exit (EXIT_FAILURE);
00067      }
00068  }
00069  else {
00070      infilep = stdin;
00071  }
00072
00073  ix = -1;
00074
00075  while (fgets (inbuf, MAXBUF-1, infilep) != NULL) {
00076      sscanf (inbuf, "%X", (unsigned *)&iy);
00077      if (ix == iy) fprintf (stderr, "Duplicate code point: %04X\n", ix);
00078      else ix = iy;
00079  }
00080  exit (0);
00081 }

```

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 Unioctde 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

Copyright © 2023 Paul Hardy

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](#).

```
00093     {
00094
00095     unsigned char tmp_byte;
00096     unsigned char mask;
00097     int row, column;
00098
00099     for (row = 0; row < 16; row++) {
00100         tmp_byte = glyph [row][0];
00101         mask = 0x80;
00102         for (column = 0; column < 8; column++) {
00103             glyphbits [row][column] = tmp_byte & mask ? 1 : 0;
00104             mask »= 1;
00105         }
00106
00107         if (width > 8)
00108             tmp_byte = glyph [row][1];
00109         else
00110             tmp_byte = 0x00;
00111
00112         mask = 0x80;
00113         for (column = 8; column < 16; column++) {
00114             glyphbits [row][column] = tmp_byte & mask ? 1 : 0;
00115             mask »= 1;
00116         }
00117     }
00118
00119     return;
00120 }
00121 }
```

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](#).

```

00223     {
00224
00225     int i;          /* index into outstring array */
00226     int row;
00227
00228     if (codept <= 0xFFFF) {
00229         sprintf (outstring, "%04X:", codept);
00230         i = 5;
00231     }
00232     else {
00233         sprintf (outstring, "%06X:", codept);
00234         i = 7;
00235     }
00236
00237     for (row = 0; row < 16; row++) {
00238         sprintf (&outstring[i], "%02X", glyph [row][0]);
00239         i += 2;
00240
00241         if (width > 8) {
00242             sprintf (&outstring[i], "%02X", glyph [row][1]);
00243             i += 2;
00244         }
00245     }
00246
00247     outstring[i] = '\0'; /* terminate output string */
00248
00249
00250     return;
00251 }

```

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 or 16 columns of 8 pixels.

Definition at line 150 of file [unifont-support.c](#).

```

00152         {
00153
00154     int column;
00155
00156
00157     for (column = 0; column < 8; column++) {
00158         transpose[0][column] =
00159             (glyphbits[0][column] << 7) |
00160             (glyphbits[1][column] << 6) |
00161             (glyphbits[2][column] << 5) |
00162             (glyphbits[3][column] << 4) |
00163             (glyphbits[4][column] << 3) |
00164             (glyphbits[5][column] << 2) |
00165             (glyphbits[6][column] << 1) |
00166             (glyphbits[7][column] );
00167         transpose[1][column] =
00168             (glyphbits[8][column] << 7) |
00169             (glyphbits[9][column] << 6) |
00170             (glyphbits[10][column] << 5) |
00171             (glyphbits[11][column] << 4) |
00172             (glyphbits[12][column] << 3) |
00173             (glyphbits[13][column] << 2) |
00174             (glyphbits[14][column] << 1) |
00175             (glyphbits[15][column] );
00176     }
00177     if (width > 8) {
00178         for (column = 8; column < width; column++) {
00179             transpose[0][column] =
00180                 (glyphbits[0][column] << 7) |
00181                 (glyphbits[1][column] << 6) |
00182                 (glyphbits[2][column] << 5) |
00183                 (glyphbits[3][column] << 4) |
00184                 (glyphbits[4][column] << 3) |
00185                 (glyphbits[5][column] << 2) |
00186                 (glyphbits[6][column] << 1) |
00187                 (glyphbits[7][column] );
00188             transpose[1][column] =
00189                 (glyphbits[8][column] << 7) |
00190                 (glyphbits[9][column] << 6) |
00191                 (glyphbits[10][column] << 5) |
00192                 (glyphbits[11][column] << 4) |
00193                 (glyphbits[12][column] << 3) |
00194                 (glyphbits[13][column] << 2) |
00195                 (glyphbits[14][column] << 1) |
00196                 (glyphbits[15][column] );
00197         }
00198     }
00199     else {
00200         for (column = 8; column < width; column++)
00201             transpose[0][column] = transpose[1][column] = 0x00;
00202     }
00203

```



```

00204
00205     return;
00206 }

```

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 Unioode 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](#).

```

00047     {
00048
00049     int i;
00050     int row;
00051     int length;
00052
00053     sscanf (hexstring, "%X", codept);
00054     length = strlen (hexstring);
00055     for (i = length - 1; i > 0 && hexstring[i] != '\n'; i--);
00056     hexstring[i] = '\0';
00057     for (i = 0; i < 9 && hexstring[i] != ':'; i++);
00058     i++; /* Skip over ':' */
00059     *width = (length - i) * 4 / 16; /* 16 rows per glyphbits */
00060
00061     for (row = 0; row < 16; row++) {
00062         sscanf (&hexstring[i], "%2hhX", &glyph [row][0]);
00063         i += 2;
00064         if (*width > 8) {
00065             sscanf (&hexstring[i], "%2hhX", &glyph [row][1]);
00066             i += 2;
00067         }
00068         else {
00069             glyph [row][1] = 0x00;
00070         }
00071     }
00072
00073
00074     return;
00075 }

```

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](#).

```
00269     {
00270
00271     int i;          /* index into outstring array */
00272     int column;
00273
00274     if (codept <= 0xFFFF) {
00275         sprintf (outstring, "%04X:", codept);
00276         i = 5;
00277     }
00278     else {
00279         sprintf (outstring, "%06X:", codept);
00280         i = 7;
00281     }
00282
00283     for (column = 0; column < 8; column++) {
00284         sprintf (&outstring[i], "%02X", transpose [0][column]);
00285         i += 2;
00286     }
00287     if (width > 8) {
00288         for (column = 8; column < 16; column++) {
00289             sprintf (&outstring[i], "%02X", transpose [0][column]);
00290             i += 2;
00291         }
00292     }
00293     for (column = 0; column < 8; column++) {
00294         sprintf (&outstring[i], "%02X", transpose [1][column]);
00295         i += 2;
00296     }
00297     if (width > 8) {
00298         for (column = 8; column < 16; column++) {
00299             sprintf (&outstring[i], "%02X", transpose [1][column]);
00300             i += 2;
00301         }
00302     }
00303
00304     outstring[i] = '\0'; /* terminate output string */
00305
00306
00307     return;
00308 }
```

5.20 unifont-support.c

[Go to the documentation of this file.](#)

```

00001 /**
00002  @file: unifont-support.c
00003
00004  @brief: Support functions for Unifont .hex files.
00005
00006  @author Paul Hardy
00007
00008  @copyright Copyright © 2023 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 #include <stdio.h>
00027 #include <stdlib.h>
00028 #include <string.h>
00029
00030
00031 /**
00032  @brief Decode a Unifont .hex file into Unicode code point and glyph.
00033
00034  This function takes one line from a Unifont .hex file and decodes
00035  it into a code point followed by a 16-row glyph array. The glyph
00036  array can be one byte (8 columns) or two bytes (16 columns).
00037
00038  @param[in] hexstring The Unicode .hex string for one code point.
00039  @param[out] width The number of columns in a glyph with 16 rows.
00040  @param[out] codept The code point, contained in the first .hex file field.
00041  @param[out] glyph The Unifont glyph, as 16 rows by 1 or 2 bytes wide.
00042 */
00043 void
00044 parse_hex (char *hexstring,
00045            int *width,
00046            unsigned *codept,
00047            unsigned char glyph[16][2]) {
00048
00049     int i;
00050     int row;
00051     int length;
00052
00053     sscanf (hexstring, "%X", codept);
00054     length = strlen (hexstring);
00055     for (i = length - 1; i > 0 && hexstring[i] != '\n'; i--);
00056     hexstring[i] = '\0';
00057     for (i = 0; i < 9 && hexstring[i] != ':'; i++);
00058     i++; /* Skip over ':' */
00059     *width = (length - i) * 4 / 16; /* 16 rows per glyphbits */
00060
00061     for (row = 0; row < 16; row++) {
00062         sscanf (&hexstring[i], "%2hhX", &glyph [row][0]);
00063         i += 2;
00064         if (*width > 8) {
00065             sscanf (&hexstring[i], "%2hhX", &glyph [row][1]);
00066             i += 2;
00067         }
00068         else {
00069             glyph [row][1] = 0x00;
00070         }
00071     }
00072
00073
00074     return;
00075 }
00076

```

```

00077
00078 /**
00079  @brief Convert a Unifont binary glyph into a binary glyph array of bits.
00080
00081  This function takes a Unifont 16-row by 1- or 2-byte wide binary glyph
00082  and returns an array of 16 rows by 16 columns. For each output array
00083  element, a 1 indicates the corresponding bit was set in the binary
00084  glyph, and a 0 indicates the corresponding bit was not set.
00085
00086  @param[in] width The number of columns in the glyph.
00087  @param[in] glyph The binary glyph, as a 16-row by 2-byte array.
00088  @param[out] glyphbits The converted glyph, as a 16-row, 16-column array.
00089 */
00090 void
00091 glyph2bits (int width,
00092             unsigned char glyph[16][2],
00093             unsigned char glyphbits [16][16]) {
00094
00095     unsigned char tmp_byte;
00096     unsigned char mask;
00097     int row, column;
00098
00099     for (row = 0; row < 16; row++) {
00100         tmp_byte = glyph [row][0];
00101         mask = 0x80;
00102         for (column = 0; column < 8; column++) {
00103             glyphbits [row][column] = tmp_byte & mask ? 1 : 0;
00104             mask »= 1;
00105         }
00106
00107         if (width > 8)
00108             tmp_byte = glyph [row][1];
00109         else
00110             tmp_byte = 0x00;
00111
00112         mask = 0x80;
00113         for (column = 8; column < 16; column++) {
00114             glyphbits [row][column] = tmp_byte & mask ? 1 : 0;
00115             mask »= 1;
00116         }
00117     }
00118
00119     return;
00120 }
00121
00122 /**
00123  @brief Transpose a Unifont .hex format glyph into 2 column-major sub-arrays.
00124
00125  This function takes a 16-by-16 cell bit array made from a Unifont
00126  glyph (as created by the glyph2bits function) and outputs a transposed
00127  array of 2 sets of 8 or 16 columns, depending on the glyph width.
00128  This format simplifies outputting these bit patterns on a graphics
00129  display with a controller chip designed to output a column of 8 pixels
00130  at a time.
00131
00132  For a line of text with Unifont output, first all glyphs can have
00133  their first 8 rows of pixels displayed on a line. Then the second
00134  8 rows of all glyphs on the line can be displayed. This simplifies
00135  code for such controller chips that are designed to automatically
00136  increment input bytes of column data by one column at a time for
00137  each successive byte.
00138
00139  The glyphbits array contains a '1' in each cell where the corresponding
00140  non-transposed glyph has a pixel set, and 0 in each cell where a pixel
00141  is not set.
00142
00143  @param[in] width The number of columns in the glyph.
00144  @param[in] glyphbits The 16-by-16 pixel glyph bits.
00145  @param[out] transpose The array of 2 sets of 8 or 16 columns of 8 pixels.
00146 */
00147 void
00148 hexpose (int width,
00149          unsigned char glyphbits [16][16],
00150          unsigned char transpose [2][16]) {
00151
00152     int column;
00153
00154     for (column = 0; column < 8; column++) {

```

```

00158     transpose [0][column] =
00159         (glyphbits [ 0][column] « 7) |
00160         (glyphbits [ 1][column] « 6) |
00161         (glyphbits [ 2][column] « 5) |
00162         (glyphbits [ 3][column] « 4) |
00163         (glyphbits [ 4][column] « 3) |
00164         (glyphbits [ 5][column] « 2) |
00165         (glyphbits [ 6][column] « 1) |
00166         (glyphbits [ 7][column] );
00167     transpose [1][column] =
00168         (glyphbits [ 8][column] « 7) |
00169         (glyphbits [ 9][column] « 6) |
00170         (glyphbits [10][column] « 5) |
00171         (glyphbits [11][column] « 4) |
00172         (glyphbits [12][column] « 3) |
00173         (glyphbits [13][column] « 2) |
00174         (glyphbits [14][column] « 1) |
00175         (glyphbits [15][column] );
00176 }
00177 if (width > 8) {
00178     for (column = 8; column < width; column++) {
00179         transpose [0][column] =
00180             (glyphbits [0][column] « 7) |
00181             (glyphbits [1][column] « 6) |
00182             (glyphbits [2][column] « 5) |
00183             (glyphbits [3][column] « 4) |
00184             (glyphbits [4][column] « 3) |
00185             (glyphbits [5][column] « 2) |
00186             (glyphbits [6][column] « 1) |
00187             (glyphbits [7][column] );
00188         transpose [1][column] =
00189             (glyphbits [ 8][column] « 7) |
00190             (glyphbits [ 9][column] « 6) |
00191             (glyphbits [10][column] « 5) |
00192             (glyphbits [11][column] « 4) |
00193             (glyphbits [12][column] « 3) |
00194             (glyphbits [13][column] « 2) |
00195             (glyphbits [14][column] « 1) |
00196             (glyphbits [15][column] );
00197     }
00198 }
00199 else {
00200     for (column = 8; column < width; column++)
00201         transpose [0][column] = transpose [1][column] = 0x00;
00202 }
00203
00204 return;
00205 }
00206 }
00207
00208 /**
00209  @brief Convert a glyph code point and byte array into a Unifont .hex string.
00210
00211  This function takes a code point and a 16-row by 1- or 2-byte binary
00212  glyph, and converts it into a Unifont .hex format character array.
00213
00214  @param[in] width The number of columns in the glyph.
00215  @param[in] codept The code point to appear in the output .hex string.
00216  @param[in] glyph The glyph, with each of 16 rows 1 or 2 bytes wide.
00217  @param[out] outstring The output string, in Unifont .hex format.
00218 */
00219 void
00220 glyph2string (int width, unsigned codept,
00221              unsigned char glyph [16][2],
00222              char *outstring) {
00223     int i; /* index into outstring array */
00224     int row;
00225     if (codept <= 0xFFFF) {
00226         sprintf (outstring, "%04X:", codept);
00227         i = 5;
00228     }
00229     else {
00230         sprintf (outstring, "%06X:", codept);
00231         i = 7;
00232     }
00233     for (row = 0; row < 16; row++) {
00234         sprintf (&outstring[i], "%02X", glyph [row][0]);

```

```

00239     i += 2;
00240
00241     if (width > 8) {
00242         sprintf (&outstring[i], "%02X", glyph [row][1]);
00243         i += 2;
00244     }
00245 }
00246
00247 outstring[i] = '\\0'; /* terminate output string */
00248
00249
00250 return;
00251 }
00252
00253
00254 /**
00255  @brief Convert a code point and transposed glyph into a Unifont .hex string.
00256
00257  This function takes a code point and a transposed Unifont glyph
00258  of 2 rows of 8 pixels in a column, and converts it into a Unifont
00259  .hex format character array.
00260
00261  @param[in] width The number of columns in the glyph.
00262  @param[in] codept The code point to appear in the output .hex string.
00263  @param[in] transpose The transposed glyph, with 2 sets of 8-row data.
00264  @param[out] outstring The output string, in Unifont .hex format.
00265 */
00266 void
00267 xglyph2string (int width, unsigned codept,
00268               unsigned char transpose [2][16],
00269               char *outstring) {
00270
00271     int i;          /* index into outstring array */
00272     int column;
00273
00274     if (codept <= 0xFFFF) {
00275         sprintf (outstring, "%04X:", codept);
00276         i = 5;
00277     }
00278     else {
00279         sprintf (outstring, "%06X:", codept);
00280         i = 7;
00281     }
00282
00283     for (column = 0; column < 8; column++) {
00284         sprintf (&outstring[i], "%02X", transpose [0][column]);
00285         i += 2;
00286     }
00287     if (width > 8) {
00288         for (column = 8; column < 16; column++) {
00289             sprintf (&outstring[i], "%02X", transpose [0][column]);
00290             i += 2;
00291         }
00292     }
00293     for (column = 0; column < 8; column++) {
00294         sprintf (&outstring[i], "%02X", transpose [1][column]);
00295         i += 2;
00296     }
00297     if (width > 8) {
00298         for (column = 8; column < 16; column++) {
00299             sprintf (&outstring[i], "%02X", transpose [1][column]);
00300             i += 2;
00301         }
00302     }
00303
00304     outstring[i] = '\\0'; /* terminate output string */
00305
00306     return;
00307 }
00308 }
00309

```

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](#) (void)
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](mailto:unifoundry@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 64 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 61 of file [unifont1per.c](#).

5.21.3 Function Documentation

5.21.3.1 main()

```
int main (
    void )
```

The main function.

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 73 of file [unifont1per.c](#).

```
00073     {
00074
00075     int i; /* loop variable */
00076
00077     /*
00078      * Define bitmap header bytes
00079      */
00080     unsigned char header [62] = {
00081         /*
00082          * Bitmap File Header -- 14 bytes
00083          */
00084         'B', 'M', /* Signature */
00085         0x7E, 0, 0, 0, /* File Size */
00086         0, 0, 0, 0, /* Reserved */
00087         0x3E, 0, 0, 0, /* Pixel Array Offset */
00088
00089         /*
00090          * Device Independent Bitmap Header -- 40 bytes
00091          */
00092         /* Image Width and Image Height are assigned final values
00093          * based on the dimensions of each glyph.
00094          */
```



```

00095     0x28, 0, 0, 0, /* DIB Header Size */
00096     0x10, 0, 0, 0, /* Image Width = 16 pixels */
00097     0xF0, 0xFF, 0xFF, 0xFF, /* Image Height = -16 pixels */
00098     0x01, 0, /* Planes */
00099     0x01, 0, /* Bits Per Pixel */
00100     0, 0, 0, 0, /* Compression */
00101     0x40, 0, 0, 0, /* Image Size */
00102     0x14, 0x0B, 0, 0, /* X Pixels Per Meter = 72 dpi */
00103     0x14, 0x0B, 0, 0, /* Y Pixels Per Meter = 72 dpi */
00104     0x02, 0, 0, 0, /* Colors In Color Table */
00105     0, 0, 0, 0, /* Important Colors */
00106
00107     /*
00108     Color Palette -- 8 bytes
00109     */
00110     0xFF, 0xFF, 0xFF, 0, /* White */
00111     0, 0, 0, 0 /* Black */
00112 };
00113
00114 char instring[MAXSTRING]; /* input string */
00115 unsigned code_point; /* current Unicode code point */
00116 char glyph[MAXSTRING]; /* bitmap string for this glyph */
00117 int glyph_height=16; /* for now, fixed at 16 pixels high */
00118 int glyph_width; /* 8, 16, 24, or 32 pixels wide */
00119 char filename[MAXFILENAME]; /* name of current output file */
00120 FILE *outfp; /* file pointer to current output file */
00121
00122 int string_index; /* pointer into hexadecimal glyph string */
00123 unsigned nextbyte; /* next set of 8 bits to print out */
00124
00125 /* Repeat for each line in the input stream */
00126 while (fgets (instring, MAXSTRING - 1, stdin) != NULL) {
00127     /* Read next Unifont ASCII hexadecimal format glyph description */
00128     sscanf (instring, "%X:%s", &code_point, glyph);
00129     /* Calculate width of a glyph in pixels; 4 bits per ASCII hex digit */
00130     glyph_width = strlen (glyph) / (glyph_height / 4);
00131     snprintf (filename, MAXFILENAME, "U+%06X.bmp", code_point);
00132     header [18] = glyph_width; /* bitmap width */
00133     header [22] = -glyph_height; /* negative height --> draw top to bottom */
00134     if ((outfp = fopen (filename, "w")) != NULL) {
00135         for (i = 0; i < 62; i++) fputc (header[i], outfp);
00136         /*
00137         Bitmap, with each row padded with zeroes if necessary
00138         so each row is four bytes wide. (Each row must end
00139         on a four-byte boundary, and four bytes is the maximum
00140         possible row length for up to 32 pixels in a row.)
00141         */
00142         string_index = 0;
00143         for (i = 0; i < glyph_height; i++) {
00144             /* Read 2 ASCII hexadecimal digits (1 byte of output pixels) */
00145             sscanf (&glyph[string_index], "%2X", &nextbyte);
00146             string_index += 2;
00147             fputc (nextbyte, outfp); /* write out the 8 pixels */
00148             if (glyph_width <= 8) { /* pad row with 3 zero bytes */
00149                 fputc (0x00, outfp); fputc (0x00, outfp); fputc (0x00, outfp);
00150             }
00151             else { /* get 8 more pixels */
00152                 sscanf (&glyph[string_index], "%2X", &nextbyte);
00153                 string_index += 2;
00154                 fputc (nextbyte, outfp); /* write out the 8 pixels */
00155                 if (glyph_width <= 16) { /* pad row with 2 zero bytes */
00156                     fputc (0x00, outfp); fputc (0x00, outfp);
00157                 }
00158                 else { /* get 8 more pixels */
00159                     sscanf (&glyph[string_index], "%2X", &nextbyte);
00160                     string_index += 2;
00161                     fputc (nextbyte, outfp); /* write out the 8 pixels */
00162                     if (glyph_width <= 24) { /* pad row with 1 zero byte */
00163                         fputc (0x00, outfp);
00164                     }
00165                     else { /* get 8 more pixels */
00166                         sscanf (&glyph[string_index], "%2X", &nextbyte);
00167                         string_index += 2;
00168                         fputc (nextbyte, outfp); /* write out the 8 pixels */
00169                     } /* glyph is 32 pixels wide */
00170                 } /* glyph is 24 pixels wide */
00171             } /* glyph is 16 pixels wide */
00172         } /* glyph is 8 pixels wide */
00173
00174         fclose (outfp);
00175     }

```

```

00176 }
00177
00178     exit (EXIT_SUCCESS);
00179 }

```

5.22 unifont1per.c

[Go to the documentation of this file.](#)

```

00001 /**
00002     @file unifont1per.c
00003
00004     @brief unifont1per - Read a Unifont .hex file from standard input and
00005             produce one glyph per ".bmp" bitmap file as output
00006
00007     @author Paul Hardy, unifoundry <at> unifoundry.com, December 2016
00008
00009     @copyright Copyright (C) 2016, 2017 Paul Hardy
00010
00011     Each glyph is 16 pixels tall, and can be 8, 16, 24,
00012     or 32 pixels wide. The width of each output graphic
00013     file is determined automatically by the width of each
00014     Unifont hex representation.
00015
00016     This program creates files of the form "U+<codepoint>.bmp", 1 per glyph.
00017
00018     Synopsis: unifont1per < unifont.hex
00019 */
00020 /*
00021     LICENSE:
00022
00023     This program is free software: you can redistribute it and/or modify
00024     it under the terms of the GNU General Public License as published by
00025     the Free Software Foundation, either version 2 of the License, or
00026     (at your option) any later version.
00027
00028     This program is distributed in the hope that it will be useful,
00029     but WITHOUT ANY WARRANTY; without even the implied warranty of
00030     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00031     GNU General Public License for more details.
00032
00033     You should have received a copy of the GNU General Public License
00034     along with this program. If not, see <http://www.gnu.org/licenses/>.
00035
00036     Example:
00037
00038     mkdir my-bmp
00039     cd my-bmp
00040     unifont1per < ../glyphs.hex
00041 */
00042 */
00043
00044 /*
00045     11 May 2019 [Paul Hardy]:
00046     - Changed sprintf function call to snprintf for writing
00047     "filename" character string.
00048     - Defined MAXFILENAME to hold size of "filename" array
00049     for snprintf function call.
00050
00051     6 September 2025 [Paul Hardy]:
00052     - Changed code_point and nextbyte from "int" to "unsigned" for
00053     compatibility with sscanf definition.
00054 */
00055
00056 #include <stdio.h>
00057 #include <stdlib.h>
00058 #include <string.h>
00059
00060 /** Maximum size of an input line in a Unifont .hex file - 1. */
00061 #define MAXSTRING 266
00062
00063 /** Maximum size of a filename of the form "U+%06X.bmp". */
00064 #define MAXFILENAME 20
00065
00066
00067 /**

```

```

00068  @brief The main function.
00069
00070  @return This program exits with status EXIT_SUCCESS.
00071  */
00072  int
00073  main (void) {
00074
00075      int i; /* loop variable */
00076
00077      /*
00078       * Define bitmap header bytes
00079       */
00080      unsigned char header [62] = {
00081          /*
00082           * Bitmap File Header -- 14 bytes
00083           */
00084          'B', 'M', /* Signature */
00085          0x7E, 0, 0, 0, /* File Size */
00086          0, 0, 0, 0, /* Reserved */
00087          0x3E, 0, 0, 0, /* Pixel Array Offset */
00088
00089          /*
00090           * Device Independent Bitmap Header -- 40 bytes
00091           */
00092          /* Image Width and Image Height are assigned final values
00093           * based on the dimensions of each glyph.
00094           */
00095          0x28, 0, 0, 0, /* DIB Header Size */
00096          0x10, 0, 0, 0, /* Image Width = 16 pixels */
00097          0xF0, 0xFF, 0xFF, 0xFF, /* Image Height = -16 pixels */
00098          0x01, 0, /* Planes */
00099          0x01, 0, /* Bits Per Pixel */
00100          0, 0, 0, 0, /* Compression */
00101          0x40, 0, 0, 0, /* Image Size */
00102          0x14, 0x0B, 0, 0, /* X Pixels Per Meter = 72 dpi */
00103          0x14, 0x0B, 0, 0, /* Y Pixels Per Meter = 72 dpi */
00104          0x02, 0, 0, 0, /* Colors In Color Table */
00105          0, 0, 0, 0, /* Important Colors */
00106
00107          /*
00108           * Color Palette -- 8 bytes
00109           */
00110          0xFF, 0xFF, 0xFF, 0, /* White */
00111          0, 0, 0, 0 /* Black */
00112      };
00113
00114      char instring[MAXSTRING]; /* input string */
00115      unsigned code_point; /* current Unicode code point */
00116      char glyph[MAXSTRING]; /* bitmap string for this glyph */
00117      int glyph_height=16; /* for now, fixed at 16 pixels high */
00118      int glyph_width; /* 8, 16, 24, or 32 pixels wide */
00119      char filename[MAXFILENAME]; /* name of current output file */
00120      FILE *outfp; /* file pointer to current output file */
00121
00122      int string_index; /* pointer into hexadecimal glyph string */
00123      unsigned nextbyte; /* next set of 8 bits to print out */
00124
00125      /* Repeat for each line in the input stream */
00126      while (fgets (instring, MAXSTRING - 1, stdin) != NULL) {
00127          /* Read next Unifont ASCII hexadecimal format glyph description */
00128          sscanf (instring, "%X:%s", &code_point, glyph);
00129          /* Calculate width of a glyph in pixels; 4 bits per ASCII hex digit */
00130          glyph_width = strlen (glyph) / (glyph_height / 4);
00131          snprintf (filename, MAXFILENAME, "U+%06X.bmp", code_point);
00132          header [18] = glyph_width; /* bitmap width */
00133          header [22] = -glyph_height; /* negative height --> draw top to bottom */
00134          if ((outfp = fopen (filename, "w")) != NULL) {
00135              for (i = 0; i < 62; i++) fputc (header[i], outfp);
00136              /*
00137               * Bitmap, with each row padded with zeroes if necessary
00138               * so each row is four bytes wide. (Each row must end
00139               * on a four-byte boundary, and four bytes is the maximum
00140               * possible row length for up to 32 pixels in a row.)
00141               */
00142              string_index = 0;
00143              for (i = 0; i < glyph_height; i++) {
00144                  /* Read 2 ASCII hexadecimal digits (1 byte of output pixels) */
00145                  sscanf (&glyph[string_index], "%2X", &nextbyte);
00146                  string_index += 2;
00147                  fputc (nextbyte, outfp); /* write out the 8 pixels */
00148                  if (glyph_width <= 8) { /* pad row with 3 zero bytes */

```

```

00149         fputc (0x00, outfp); fputc (0x00, outfp); fputc (0x00, outfp);
00150     }
00151     else { /* get 8 more pixels */
00152         sscanf (&glyph[string_index], "%2X", &nextbyte);
00153         string_index += 2;
00154         fputc (nextbyte, outfp); /* write out the 8 pixels */
00155         if (glyph_width <= 16) { /* pad row with 2 zero bytes */
00156             fputc (0x00, outfp); fputc (0x00, outfp);
00157         }
00158         else { /* get 8 more pixels */
00159             sscanf (&glyph[string_index], "%2X", &nextbyte);
00160             string_index += 2;
00161             fputc (nextbyte, outfp); /* write out the 8 pixels */
00162             if (glyph_width <= 24) { /* pad row with 1 zero byte */
00163                 fputc (0x00, outfp);
00164             }
00165             else { /* get 8 more pixels */
00166                 sscanf (&glyph[string_index], "%2X", &nextbyte);
00167                 string_index += 2;
00168                 fputc (nextbyte, outfp); /* write out the 8 pixels */
00169             } /* glyph is 32 pixels wide */
00170         } /* glyph is 24 pixels wide */
00171     } /* glyph is 16 pixels wide */
00172 } /* glyph is 8 pixels wide */
00173
00174     fclose (outfp);
00175 }
00176 }
00177
00178     exit (EXIT_SUCCESS);
00179 }

```

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 [78](#) 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 308 of file [unifontpic.c](#).

```

00309 {
00310
00311     char header_string[HDR_LEN]; /* centered header */
00312     char raw_header[HDR_LEN]; /* left-aligned header */
00313     int header[16][16]; /* header row, for chart title */
00314     int hdrlen; /* length of HEADER_STRING */
00315     int startcol; /* column to start printing header, for centering */
00316
00317     unsigned leftcol[0x1000][16]; /* code point legend on left side of chart */
00318     int d1, d2, d3, d4; /* digits for filling leftcol[][] legend */
00319     int codept; /* current starting code point for legend */
00320     int thisrow; /* glyph row currently being rendered */
00321     unsigned toprow[16][16]; /* code point legend on top of chart */
00322     int digitrow; /* row we're in (0..4) for the above hexdigit digits */
00323
00324     /*
00325      * DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
00326      */
00327     int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
00328     int ImageSize;
00329     int FileSize;
00330     int Width, Height; /* bitmap image width and height in pixels */
00331     int ppm; /* integer pixels per meter */
00332
00333     int i, j, k;
00334
00335     unsigned bytesout;
00336
00337     void output4(int), output2(int);
00338
00339     /*
00340      * Image width and height, in pixels.
00341
00342      * N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
00343      */
00344     Width = 18 * 16; /* (2 legend + 16 glyphs) * 16 pixels/glyph */
00345     Height = 4099 * 16; /* (1 header + 4096 glyphs) * 16 rows/glyph */
00346
00347     ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */
00348
00349     FileSize = DataOffset + ImageSize;
00350
00351     /* convert dots/inch to pixels/meter */
00352     if (dpi == 0) dpi = 96;
00353     ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);
00354
00355     /*
00356      * Generate the BMP Header
00357      */
00358     putchar ('B');
00359     putchar ('M');
00360
00361     /*
00362      * Calculate file size:
00363
00364      * BMP Header + InfoHeader + Color Table + Raster Data
00365      */
00366     output4 (FileSize); /* FileSize */
00367     output4 (0x0000); /* reserved */
00368
00369     /* Calculate DataOffset */
00370     output4 (DataOffset);
00371
00372     /*
00373      * InfoHeader
00374      */

```

```

00375 output4 (40);          /* Size of InfoHeader */
00376 output4 (Width);       /* Width of bitmap in pixels */
00377 output4 (Height);      /* Height of bitmap in pixels */
00378 output2 (1);          /* Planes (1 plane) */
00379 output2 (1);          /* BitCount (1 = monochrome) */
00380 output4 (0);          /* Compression (0 = none) */
00381 output4 (ImageSize);   /* ImageSize, in bytes */
00382 output4 (ppm);         /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
00383 output4 (ppm);         /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
00384 output4 (2);          /* ColorsUsed (= 2) */
00385 output4 (2);          /* ColorsImportant (= 2) */
00386 output4 (0x00000000); /* black (reserved, B, G, R) */
00387 output4 (0x00FFFFFF); /* white (reserved, B, G, R) */
00388
00389 /*
00390  * Create header row bits.
00391  */
00392 snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
00393 memset ((void *)header, 0, 16 * 16 * sizeof (int)); /* fill with white */
00394 memset ((void *)header_string, ' ', 32 * sizeof (char)); /* 32 spaces */
00395 header_string[32] = '\0'; /* null-terminated */
00396
00397 hdrlen = strlen (raw_header);
00398 if (hdrlen > 32) hdrlen = 32; /* only 32 columns to print header */
00399 startcol = 16 - ((hdrlen + 1) » 1); /* to center header */
00400 /* center up to 32 chars */
00401 memcpy (&header_string[startcol], raw_header, hdrlen);
00402
00403 /* Copy each letter's bitmap from the plane_array[] we constructed. */
00404 /* Each glyph must be single-width, to fit two glyphs in 16 pixels */
00405 for (j = 0; j < 16; j++) {
00406     for (i = 0; i < 16; i++) {
00407         header[i][j] =
00408             (ascii_bits[header_string[j+j ] & 0x7F][i] & 0xFF00) |
00409             (ascii_bits[header_string[j+j+1] & 0x7F][i] » 8);
00410     }
00411 }
00412
00413 /*
00414  * Create the left column legend.
00415  */
00416 memset ((void *)leftcol, 0, 4096 * 16 * sizeof (unsigned));
00417
00418 for (codept = 0x0000; codept < 0x10000; codept += 0x10) {
00419     d1 = (codept » 12) & 0xF; /* most significant hex digit */
00420     d2 = (codept » 8) & 0xF;
00421     d3 = (codept » 4) & 0xF;
00422
00423     thisrow = codept » 4; /* rows of 16 glyphs */
00424
00425     /* fill in first and second digits */
00426     for (digitrow = 0; digitrow < 5; digitrow++) {
00427         leftcol[thisrow][2 + digitrow] =
00428             (hexdigit[d1][digitrow] « 10) |
00429             (hexdigit[d2][digitrow] « 4);
00430     }
00431
00432     /* fill in third digit */
00433     for (digitrow = 0; digitrow < 5; digitrow++) {
00434         leftcol[thisrow][9 + digitrow] = hexdigit[d3][digitrow] « 10;
00435     }
00436     leftcol[thisrow][9 + 4] |= 0xF « 4; /* underscore as 4th digit */
00437
00438     for (i = 0; i < 15; i++) {
00439         leftcol[thisrow][i] |= 0x00000002; /* right border */
00440     }
00441
00442     leftcol[thisrow][15] = 0x0000FFFE; /* bottom border */
00443
00444     if (d3 == 0xF) { /* 256-point boundary */
00445         leftcol[thisrow][15] |= 0x00FF0000; /* longer tic mark */
00446     }
00447
00448     if ((thisrow % 0x40) == 0x3F) { /* 1024-point boundary */
00449         leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
00450     }
00451 }
00452
00453 /*
00454  * Create the top row legend.
00455  */

```

```

00456  memset ((void *)toprow, 0, 16 * 16 * sizeof (unsigned));
00457
00458  for (codept = 0x0; codept <= 0xF; codept++) {
00459      d1 = (codept » 12) & 0xF; /* most significant hex digit */
00460      d2 = (codept » 8) & 0xF;
00461      d3 = (codept » 4) & 0xF;
00462      d4 = codept & 0xF; /* least significant hex digit */
00463
00464      /* fill in last digit */
00465      for (digitrow = 0; digitrow < 5; digitrow++) {
00466          toprow[6 + digitrow][codept] = hexdigit[d4][digitrow] « 6;
00467      }
00468  }
00469
00470  for (j = 0; j < 16; j++) {
00471      /* force bottom pixel row to be white, for separation from glyphs */
00472      toprow[15][j] = 0x0000;
00473  }
00474
00475  /* 1 pixel row with left-hand legend line */
00476  for (j = 0; j < 16; j++) {
00477      toprow[14][j] |= 0xFFFF;
00478  }
00479
00480  /* 14 rows with line on left to fill out this character row */
00481  for (i = 13; i >= 0; i--) {
00482      for (j = 0; j < 16; j++) {
00483          toprow[i][j] |= 0x0001;
00484      }
00485  }
00486
00487  /*
00488   Now write the raster image.
00489
00490   XOR each byte with 0xFF because black = 0, white = 1 in BMP.
00491  */
00492
00493  /* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
00494  for (i = 0xFFFF0; i >= 0; i -= 0x10) {
00495      thisrow = i » 4; /* 16 glyphs per row */
00496      for (j = 15; j >= 0; j--) {
00497          /* left-hand legend */
00498          putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
00499          putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
00500          putchar ((~leftcol[thisrow][j] » 8) & 0xFF);
00501          putchar (~leftcol[thisrow][j] & 0xFF);
00502          /* Unifont glyph */
00503          for (k = 0; k < 16; k++) {
00504              bytesout = ~plane_array[i+k][j] & 0xFFFF;
00505              putchar ((bytesout » 8) & 0xFF);
00506              putchar (~bytesout & 0xFF);
00507          }
00508      }
00509  }
00510
00511  /*
00512   Write the top legend.
00513  */
00514  /* i == 15: bottom pixel row of header is output here */
00515  /* left-hand legend: solid black line except for right-most pixel */
00516  putchar (0x00);
00517  putchar (0x00);
00518  putchar (0x00);
00519  putchar (0x01);
00520  for (j = 0; j < 16; j++) {
00521      putchar ((~toprow[15][j] » 8) & 0xFF);
00522      putchar (~toprow[15][j] & 0xFF);
00523  }
00524
00525  putchar (0xFF);
00526  putchar (0xFF);
00527  putchar (0xFF);
00528  putchar (0xFC);
00529  for (j = 0; j < 16; j++) {
00530      putchar ((~toprow[14][j] » 8) & 0xFF);
00531      putchar (~toprow[14][j] & 0xFF);
00532  }
00533
00534  for (i = 13; i >= 0; i--) {
00535      putchar (0xFF);
00536      putchar (0xFF);

```



```

00537     putchar (0xFF);
00538     putchar (0xFD);
00539     for (j = 0; j < 16; j++) {
00540         putchar ((~toprow[i][j] » 8) & 0xFF);
00541         putchar ( ~toprow[i][j]      & 0xFF);
00542     }
00543 }
00544
00545 /*
00546  Write the header.
00547 */
00548
00549 /* 7 completely white rows */
00550 for (i = 7; i >= 0; i--) {
00551     for (j = 0; j < 18; j++) {
00552         putchar (0xFF);
00553         putchar (0xFF);
00554     }
00555 }
00556
00557 for (i = 15; i >= 0; i--) {
00558     /* left-hand legend */
00559     putchar (0xFF);
00560     putchar (0xFF);
00561     putchar (0xFF);
00562     putchar (0xFF);
00563     /* header glyph */
00564     for (j = 0; j < 16; j++) {
00565         bytesout = ~header[i][j] & 0xFFFF;
00566         putchar ((bytesout » 8) & 0xFF);
00567         putchar ( bytesout      & 0xFF);
00568     }
00569 }
00570
00571 /* 8 completely white rows at very top */
00572 for (i = 7; i >= 0; i--) {
00573     for (j = 0; j < 18; j++) {
00574         putchar (0xFF);
00575         putchar (0xFF);
00576     }
00577 }
00578
00579 return;
00580 }

```

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 595 of file [unifontpic.c](#).

```

00596 {
00597
00598     char header_string[257];
00599     char raw_header[HDR_LEN];
00600     int header[16][256]; /* header row, for chart title */
00601     int hdrlen;          /* length of HEADER_STRING */
00602     int startcol;        /* column to start printing header, for centering */
00603
00604     unsigned leftcol[0x100][16]; /* code point legend on left side of chart */
00605     int d1, d2, d3, d4;          /* digits for filling leftcol[][] legend */
00606     int codept;                  /* current starting code point for legend */
00607     int thisrow;                 /* glyph row currently being rendered */
00608     unsigned toprow[32][256];   /* code point legend on top of chart */
00609     int digitrow;               /* row we're in (0..4) for the above hexdigit digits */
00610     int hexalpha1, hexalpha2;   /* to convert hex digits to ASCII */
00611
00612     /*
00613      * DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
00614      */
00615     int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
00616     int ImageSize;
00617     int FileSize;
00618     int Width, Height; /* bitmap image width and height in pixels */
00619     int ppm;           /* integer pixels per meter */
00620
00621     int i, j, k;
00622
00623     unsigned bytesout;
00624
00625     void output4(int), output2(int);
00626
00627     /*
00628      * Image width and height, in pixels.
00629      *
00630      * N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
00631      */
00632     Width = 258 * 16; /* ( 2 legend + 256 glyphs) * 16 pixels/glyph */
00633     Height = 260 * 16; /* (2 header + 2 legend + 256 glyphs) * 16 rows/glyph */
00634
00635     ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */
00636
00637     FileSize = DataOffset + ImageSize;
00638
00639     /* convert dots/inch to pixels/meter */
00640     if (dpi == 0) dpi = 96;
00641     ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);
00642
00643     /*
00644      * Generate the BMP Header
00645      */
00646     putchar ('B');
00647     putchar ('M');
00648     /*
00649      * Calculate file size:
00650      *
00651      * BMP Header + InfoHeader + Color Table + Raster Data
00652      */
00653     output4 (FileSize); /* FileSize */
00654     output4 (0x0000); /* reserved */
00655     /* Calculate DataOffset */
00656     output4 (DataOffset);
00657
00658     /*
00659      * InfoHeader
00660      */
00661     output4 (40); /* Size of InfoHeader */
00662     output4 (Width); /* Width of bitmap in pixels */
00663     output4 (Height); /* Height of bitmap in pixels */
00664     output2 (1); /* Planes (1 plane) */
00665     output2 (1); /* BitCount (1 = monochrome) */
00666     output4 (0); /* Compression (0 = none) */
00667     output4 (ImageSize); /* ImageSize, in bytes */
00668     output4 (ppm); /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
00669     output4 (ppm); /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
00670     output4 (2); /* ColorsUsed (= 2) */
00671     output4 (2); /* ColorsImportant (= 2) */
00672     output4 (0x00000000); /* black (reserved, B, G, R) */
00673     output4 (0x00FFFFFF); /* white (reserved, B, G, R) */
00674
00675     /*

```

```

00676     Create header row bits.
00677 */
00678 snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
00679 memset ((void *)header, 0, 256 * 16 * sizeof (int)); /* fill with white */
00680 memset ((void *)header_string, ' ', 256 * sizeof (char)); /* 256 spaces */
00681 header_string[256] = '\0'; /* null-terminated */
00682
00683 hdrlen = strlen (raw_header);
00684 /* Wide bitmap can print 256 columns, but limit to 32 columns for long bitmap. */
00685 if (hdrlen > 32) hdrlen = 32;
00686 startcol = 127 - ((hdrlen - 1) » 1); /* to center header */
00687 /* center up to 32 chars */
00688 memcpy (&header_string[startcol], raw_header, hdrlen);
00689
00690 /* Copy each letter's bitmap from the plane_array[] we constructed. */
00691 for (j = 0; j < 256; j++) {
00692     for (i = 0; i < 16; i++) {
00693         header[i][j] = ascii_bits[header_string[j] & 0x7F][i];
00694     }
00695 }
00696
00697 /*
00698     Create the left column legend.
00699 */
00700 memset ((void *)leftcol, 0, 256 * 16 * sizeof (unsigned));
00701
00702 for (codept = 0x0000; codept < 0x10000; codept += 0x100) {
00703     d1 = (codept » 12) & 0xF; /* most significant hex digit */
00704     d2 = (codept » 8) & 0xF;
00705
00706     thisrow = codept » 8; /* rows of 256 glyphs */
00707
00708     /* fill in first and second digits */
00709
00710     if (tinynum) { /* use 4x5 pixel glyphs */
00711         for (digitrow = 0; digitrow < 5; digitrow++) {
00712             leftcol[thisrow][6 + digitrow] =
00713                 (hexdigit[d1][digitrow] « 10) |
00714                 (hexdigit[d2][digitrow] « 4);
00715         }
00716     }
00717     else { /* bigger numbers -- use glyphs from Unifont itself */
00718         /* convert hexadecimal digits to ASCII equivalent */
00719         hexalpha1 = d1 < 0xA ? '0' + d1 : 'A' + d1 - 0xA;
00720         hexalpha2 = d2 < 0xA ? '0' + d2 : 'A' + d2 - 0xA;
00721
00722         for (i = 0; i < 16; i++) {
00723             leftcol[thisrow][i] =
00724                 (ascii_bits[hexalpha1][i] « 2) |
00725                 (ascii_bits[hexalpha2][i] » 6);
00726         }
00727     }
00728
00729     for (i = 0; i < 15; i++) {
00730         leftcol[thisrow][i] |= 0x00000002; /* right border */
00731     }
00732
00733     leftcol[thisrow][15] = 0x0000FFFE; /* bottom border */
00734
00735     if (d2 == 0xF) { /* 4096-point boundary */
00736         leftcol[thisrow][15] |= 0x00FF0000; /* longer tic mark */
00737     }
00738
00739     if ((thisrow % 0x40) == 0x3F) { /* 16,384-point boundary */
00740         leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
00741     }
00742 }
00743
00744 /*
00745     Create the top row legend.
00746 */
00747 memset ((void *)toprow, 0, 32 * 256 * sizeof (unsigned));
00748
00749 for (codept = 0x00; codept <= 0xFF; codept++) {
00750     d3 = (codept » 4) & 0xF;
00751     d4 = codept & 0xF; /* least significant hex digit */
00752
00753     if (tinynum) {
00754         for (digitrow = 0; digitrow < 5; digitrow++) {
00755             toprow[16 + 6 + digitrow][codept] =
00756                 (hexdigit[d3][digitrow] « 10) |

```

```

00757         (hexdigit[d4][digitrow] « 4);
00758     }
00759 }
00760 else {
00761     /* convert hexadecimal digits to ASCII equivalent */
00762     hexalpha1 = d3 < 0xA ? '0' + d3 : 'A' + d3 - 0xA;
00763     hexalpha2 = d4 < 0xA ? '0' + d4 : 'A' + d4 - 0xA;
00764     for (i = 0; i < 16; i++) {
00765         toprow[14 + i][codept] =
00766             (ascii_bits[hexalpha1][i] ) |
00767             (ascii_bits[hexalpha2][i] » 7);
00768     }
00769 }
00770 }
00771
00772 for (j = 0; j < 256; j++) {
00773     /* force bottom pixel row to be white, for separation from glyphs */
00774     toprow[16 + 15][j] = 0x0000;
00775 }
00776
00777 /* 1 pixel row with left-hand legend line */
00778 for (j = 0; j < 256; j++) {
00779     toprow[16 + 14][j] |= 0xFFFF;
00780 }
00781
00782 /* 14 rows with line on left to fill out this character row */
00783 for (i = 13; i >= 0; i--) {
00784     for (j = 0; j < 256; j++) {
00785         toprow[16 + i][j] |= 0x0001;
00786     }
00787 }
00788
00789 /* Form the longer tic marks in top legend */
00790 for (i = 8; i < 16; i++) {
00791     for (j = 0x0F; j < 0x100; j += 0x10) {
00792         toprow[i][j] |= 0x0001;
00793     }
00794 }
00795
00796 /*
00797     Now write the raster image.
00798
00799     XOR each byte with 0xFF because black = 0, white = 1 in BMP.
00800 */
00801
00802 /* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
00803 for (i = 0xFF00; i >= 0; i -= 0x100) {
00804     thisrow = i » 8; /* 256 glyphs per row */
00805     for (j = 15; j >= 0; j--) {
00806         /* left-hand legend */
00807         putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
00808         putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
00809         putchar ((~leftcol[thisrow][j] » 8) & 0xFF);
00810         putchar ( ~leftcol[thisrow][j]      & 0xFF);
00811         /* Unifont glyph */
00812         for (k = 0x00; k < 0x100; k++) {
00813             bytesout = ~plane_array[i+k][j] & 0xFFFF;
00814             putchar ((bytesout » 8) & 0xFF);
00815             putchar ( bytesout      & 0xFF);
00816         }
00817     }
00818 }
00819
00820 /*
00821     Write the top legend.
00822 */
00823 /* i == 15: bottom pixel row of header is output here */
00824 /* left-hand legend: solid black line except for right-most pixel */
00825 putchar (0x00);
00826 putchar (0x00);
00827 putchar (0x00);
00828 putchar (0x01);
00829 for (j = 0; j < 256; j++) {
00830     putchar ((~toprow[16 + 15][j] » 8) & 0xFF);
00831     putchar ( ~toprow[16 + 15][j]      & 0xFF);
00832 }
00833
00834 putchar (0xFF);
00835 putchar (0xFF);
00836 putchar (0xFF);
00837 putchar (0xFC);

```

```

00838     for (j = 0; j < 256; j++) {
00839         putchar ((~toprow[16 + 14][j] » 8) & 0xFF);
00840         putchar ( ~toprow[16 + 14][j]      & 0xFF);
00841     }
00842
00843     for (i = 16 + 13; i >= 0; i--) {
00844         if (i >= 8) { /* make vertical stroke on right */
00845             putchar (0xFF);
00846             putchar (0xFF);
00847             putchar (0xFF);
00848             putchar (0xFD);
00849         }
00850         else { /* all white */
00851             putchar (0xFF);
00852             putchar (0xFF);
00853             putchar (0xFF);
00854             putchar (0xFF);
00855         }
00856         for (j = 0; j < 256; j++) {
00857             putchar ((~toprow[i][j] » 8) & 0xFF);
00858             putchar ( ~toprow[i][j]      & 0xFF);
00859         }
00860     }
00861
00862     /*
00863     Write the header.
00864     */
00865
00866     /* 8 completely white rows */
00867     for (i = 7; i >= 0; i--) {
00868         for (j = 0; j < 258; j++) {
00869             putchar (0xFF);
00870             putchar (0xFF);
00871         }
00872     }
00873
00874     for (i = 15; i >= 0; i--) {
00875         /* left-hand legend */
00876         putchar (0xFF);
00877         putchar (0xFF);
00878         putchar (0xFF);
00879         putchar (0xFF);
00880         /* header glyph */
00881         for (j = 0; j < 256; j++) {
00882             bytesout = ~header[i][j] & 0xFFFF;
00883             putchar ((bytesout » 8) & 0xFF);
00884             putchar ( bytesout      & 0xFF);
00885         }
00886     }
00887
00888     /* 8 completely white rows at very top */
00889     for (i = 7; i >= 0; i--) {
00890         for (j = 0; j < 258; j++) {
00891             putchar (0xFF);
00892             putchar (0xFF);
00893         }
00894     }
00895
00896     return;
00897 }

```

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 229 of file `unifontpic.c`.

```

00230 {
00231     char *bitstring; /* pointer into instring for glyph bitmap */
00232     int i; /* loop variable */
00233     unsigned codept; /* the Unicode code point of the current glyph */
00234     int glyph_plane; /* Unicode plane of current glyph */
00235     int ndigits; /* number of ASCII hexadecimal digits in glyph */
00236     int bytespl; /* bytes per line of pixels in a glyph */
00237     unsigned temprow; /* 1 row of a quadruple-width glyph */
00238     int newrow; /* 1 row of double-width output pixels */
00239     unsigned bitmask; /* to mask off 2 bits of long width glyph */
00240
00241     /*
00242      * Read each input line and place its glyph into the bit array.
00243      */
00244     sscanf (instring, "%X", &codept);
00245     glyph_plane = codept » 16;
00246     if (glyph_plane == plane) {
00247         codept &= 0xFFFF; /* array index will only have 16 bit address */
00248         /* find the colon separator */
00249         for (i = 0; (i < 9) && (instring[i] != ':'); i++);
00250         i++; /* position past it */
00251         bitstring = &instring[i];
00252         ndigits = strlen (bitstring);
00253         /* don't count '\n' at end of line if present */
00254         if (bitstring[ndigits - 1] == '\n') ndigits--;
00255         bytespl = ndigits » 5; /* 16 rows per line, 2 digits per byte */
00256
00257         if (bytespl >= 1 && bytespl <= 4) {
00258             for (i = 0; i < 16; i++) { /* 16 rows per glyph */
00259                 /* Read correct number of hexadecimal digits given glyph width */
00260                 switch (bytespl) {
00261                     case 1: sscanf (bitstring, "%2X", &temprow);
00262                             bitstring += 2;
00263                             temprow «= 8; /* left-justify single-width glyph */
00264                             break;
00265                     case 2: sscanf (bitstring, "%4X", &temprow);
00266                             bitstring += 4;
00267                             break;
00268                     /* cases 3 and 4 widths will be compressed by 50% (see below) */
00269                     case 3: sscanf (bitstring, "%6X", &temprow);
00270                             bitstring += 6;
00271                             temprow «= 8; /* left-justify */
00272                             break;
00273                     case 4: sscanf (bitstring, "%8X", &temprow);
00274                             bitstring += 8;
00275                             break;
00276                 } /* switch on number of bytes per row */
00277                 /* compress glyph width by 50% if greater than double-width */
00278                 if (bytespl > 2) {
00279                     newrow = 0x0000;
00280                     /* mask off 2 bits at a time to convert each pair to 1 bit out */
00281                     for (bitmask = 0xC0000000; bitmask != 0; bitmask »= 2) {
00282                         newrow «= 1;
00283                         if ((temprow & bitmask) != 0) newrow |= 1;
00284                     }
00285                     temprow = newrow;
00286                 } /* done conditioning glyphs beyond double-width */
00287                 plane_array[codept][i] = temprow; /* store glyph bitmap for output */
00288             } /* for each row */
00289         } /* if 1 to 4 bytes per row/line */
00290     } /* if this is the plane we are seeking */
00291
00292     return;
00293 }

```

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 98 of file [unifontpic.c](#).

```
00099 {
00100     /* Input line buffer */
00101     char instring[MAXSTRING];
00102
00103     /* long and dpi are set from command-line options */
00104     int wide=1; /* =1 for a 256x256 grid, =0 for a 16x4096 grid */
00105     int dpi=96; /* change for 256x256 grid to fit paper if desired */
00106     int tinynum=0; /* whether to use tiny labels for 256x256 grid */
00107
00108     int i, j; /* loop variables */
00109
00110     int plane=0; /* Unicode plane, 0..17; Plane 0 is default */
00111     /* 16 pixel rows for each of 65,536 glyphs in a Unicode plane */
00112     int plane_array[0x10000][16];
00113
00114     void gethex (char *instring, int plane_array[0x10000][16], int plane);
00115     void genlongbmp (int plane_array[0x10000][16], int dpi, int tinynum,
00116                     int plane);
00117     void genwidebmp (int plane_array[0x10000][16], int dpi, int tinynum,
00118                     int plane);
00119
00120     if (argc > 1) {
00121         for (i = 1; i < argc; i++) {
00122             if (strncmp (argv[i], "-l", 2) == 0) { /* long display */
00123                 wide = 0;
00124             }
00125             else if (strncmp (argv[i], "-d", 2) == 0) {
00126                 dpi = atoi (&argv[i][2]); /* dots/inch specified on command line */
00127             }
00128             else if (strncmp (argv[i], "-t", 2) == 0) {
00129                 tinynum = 1;
00130             }
00131             else if (strncmp (argv[i], "-P", 2) == 0) {
00132                 /* Get Unicode plane */
00133                 for (j = 2; argv[i][j] != '\0'; j++) {
00134                     if (argv[i][j] < '0' || argv[i][j] > '9') {
00135                         fprintf (stderr,
00136                                 "ERROR: Specify Unicode plane as decimal number.\n\n");
00137                         exit (EXIT_FAILURE);
00138                     }
00139                 }
00140                 plane = atoi (&argv[i][2]); /* Unicode plane, 0..17 */
00141                 if (plane < 0 || plane > 17) {
00142                     fprintf (stderr,
00143                             "ERROR: Plane out of Unicode range [0,17].\n\n");
00144                     exit (EXIT_FAILURE);
00145                 }
00146             }
00147         }
00148     }
```

```

00149
00150
00151  /*
00152  Initialize the ASCII bitmap array for chart titles
00153  */
00154  for (i = 0; i < 128; i++) {
00155      /* convert Unifont hexadecimal string to bitmap */
00156      gethex ((char *)ascii_hex[i], plane_array, 0);
00157      for (j = 0; j < 16; j++) ascii_bits[i][j] = plane_array[i][j];
00158  }
00159
00160
00161  /*
00162  Read in the Unifont hex file to render from standard input
00163  */
00164  memset ((void *)plane_array, 0, 0x10000 * 16 * sizeof (int));
00165  while (fgets (instring, MAXSTRING, stdin) != NULL) {
00166      gethex (instring, plane_array, plane); /* read .hex input file and fill plane_array with glyph data */
00167  } /* while not EOF */
00168
00169
00170  /*
00171  Write plane_array glyph data to BMP file as wide or long bitmap.
00172  */
00173  if (wide) {
00174      genwidebmp (plane_array, dpi, tinynum, plane);
00175  }
00176  else {
00177      genlongbmp (plane_array, dpi, tinynum, plane);
00178  }
00179
00180  exit (EXIT_SUCCESS);
00181 }

```

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 208 of file [unifontpic.c](#).

```

00209 {
00210
00211     putchar ( thisword      & 0xFF);
00212     putchar ((thisword » 8) & 0xFF);
00213
00214     return;
00215 }

```

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 190 of file [unifontpic.c](#).

```
00191 {
00192
00193     putchar ( thisword      & 0xFF);
00194     putchar ((thisword » 8) & 0xFF);
00195     putchar ((thisword » 16) & 0xFF);
00196     putchar ((thisword » 24) & 0xFF);
00197
00198     return;
00199 }
```

Here is the caller graph for this function:

5.24 unifontpic.c

[Go to the documentation of this file.](#)

```
00001 /**
00002  @file unifontpic.c
00003
00004  @brief unifontpic - See the "Big Picture": the entire Unifont
00005           in one BMP bitmap
00006
00007  @author Paul Hardy, 2013
00008
00009  @copyright Copyright (C) 2013, 2017 Paul Hardy
00010 */
00011 /*
00012  LICENSE:
00013
00014  This program is free software: you can redistribute it and/or modify
00015  it under the terms of the GNU General Public License as published by
00016  the Free Software Foundation, either version 2 of the License, or
00017  (at your option) any later version.
00018
00019  This program is distributed in the hope that it will be useful,
00020  but WITHOUT ANY WARRANTY; without even the implied warranty of
00021  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00022  GNU General Public License for more details.
00023
00024  You should have received a copy of the GNU General Public License
00025  along with this program. If not, see <http://www.gnu.org/licenses/>.
00026 */
00027
00028 /*
00029  11 June 2017 [Paul Hardy]:
00030  - Modified to take glyphs that are 24 or 32 pixels wide and
00031    compress them horizontally by 50%.
00032
00033  8 July 2017 [Paul Hardy]:
00034  - Modified to print Unifont charts above Unicode Plane 0.
00035  - Adds "-P" option to specify Unicode plane in decimal,
00036    as "-P0" through "-P17". Omitting this argument uses
00037    plane 0 as the default.
00038  - Appends Unicode plane number to chart title.
00039  - Reads in "unifontpic.h", which was added mainly to
00040    store ASCII chart title glyphs in an embedded array
00041    rather than requiring these ASCII glyphs to be in
00042    the ".hex" file that is read in for the chart body
00043    (which was the case previously, when all that was
00044    able to print was Unicode place 0).
00045  - Fixes truncated header in long bitmap format, making
00046    the long chart title glyphs single-spaced. This leaves
00047    room for the Unicode plane to appear even in the narrow
00048    chart title of the "long" format chart. The wide chart
00049    title still has double-spaced ASCII glyphs.
00050  - Adjusts centering of title on long and wide charts.
```

```

00051
00052     11 May 2019 [Paul Hardy]:
00053         - Changed strncpy calls to memcpy.
00054         - Added "HDR_LEN" to define length of header string
00055           for use in snprintf function call.
00056     - Changed sprintf function calls to snprintf function
00057       calls for writing chart header string.
00058
00059     21 October 2023 [Paul Hardy]:
00060     - Added full function prototypes in main function for
00061       functions gethex, genlongbmp, and genwidebmp.
00062     - Typecast ascii_hex[i] to char * in gethex function call
00063       to avoid warning about const char * conversion.
00064
00065     6 September 2025 [Paul Hardy]:
00066     - Changed codept and temprow from "int" to "unsigned" for
00067       compatibility with sscanf definition.
00068
00069 */
00070
00071
00072 #include <stdio.h>
00073 #include <stdlib.h>
00074 #include <string.h>
00075 #include "unifontpic.h"
00076
00077 /** Define length of header string for top of chart. */
00078 #define HDR_LEN 33
00079
00080
00081 /*
00082     Stylistic Note:
00083
00084     Many variables in this program use multiple words scrunched
00085     together, with each word starting with an upper-case letter.
00086     This is only done to match the canonical field names in the
00087     Windows Bitmap Graphics spec.
00088 */
00089
00090 /**
00091     @brief The main function.
00092
00093     @param[in] argc The count of command line arguments.
00094     @param[in] argv Pointer to array of command line arguments.
00095     @return This program exits with status EXIT_SUCCESS.
00096 */
00097 int
00098 main (int argc, char **argv)
00099 {
00100     /* Input line buffer */
00101     char instring[MAXSTRING];
00102
00103     /* long and dpi are set from command-line options */
00104     int wide=1; /* =1 for a 256x256 grid, =0 for a 16x4096 grid */
00105     int dpi=96; /* change for 256x256 grid to fit paper if desired */
00106     int tinynum=0; /* whether to use tiny labels for 256x256 grid */
00107
00108     int i, j; /* loop variables */
00109
00110     int plane=0; /* Unicode plane, 0..17; Plane 0 is default */
00111     /* 16 pixel rows for each of 65,536 glyphs in a Unicode plane */
00112     int plane_array[0x10000][16];
00113
00114     void gethex (char *instring, int plane_array[0x10000][16], int plane);
00115     void genlongbmp (int plane_array[0x10000][16], int dpi, int tinynum,
00116                     int plane);
00117     void genwidebmp (int plane_array[0x10000][16], int dpi, int tinynum,
00118                     int plane);
00119
00120     if (argc > 1) {
00121         for (i = 1; i < argc; i++) {
00122             if (strcmp (argv[i], "-l", 2) == 0) { /* long display */
00123                 wide = 0;
00124             }
00125             else if (strcmp (argv[i], "-d", 2) == 0) {
00126                 dpi = atoi (&argv[i][2]); /* dots/inch specified on command line */
00127             }
00128             else if (strcmp (argv[i], "-t", 2) == 0) {
00129                 tinynum = 1;
00130             }
00131             else if (strcmp (argv[i], "-P", 2) == 0) {

```

```

00132     /* Get Unicode plane */
00133     for (j = 2; argv[i][j] != '\0'; j++) {
00134         if (argv[i][j] < '0' || argv[i][j] > '9') {
00135             fprintf (stderr,
00136                 "ERROR: Specify Unicode plane as decimal number.\n\n");
00137             exit (EXIT_FAILURE);
00138         }
00139     }
00140     plane = atoi (&argv[i][2]); /* Unicode plane, 0..17 */
00141     if (plane < 0 || plane > 17) {
00142         fprintf (stderr,
00143             "ERROR: Plane out of Unicode range [0,17].\n\n");
00144         exit (EXIT_FAILURE);
00145     }
00146 }
00147 }
00148 }
00149
00150
00151 /*
00152  Initialize the ASCII bitmap array for chart titles
00153 */
00154 for (i = 0; i < 128; i++) {
00155     /* convert Unifont hexadecimal string to bitmap */
00156     gethex ((char *)ascii_hex[i], plane_array, 0);
00157     for (j = 0; j < 16; j++) ascii_bits[i][j] = plane_array[i][j];
00158 }
00159
00160
00161 /*
00162  Read in the Unifont hex file to render from standard input
00163 */
00164 memset ((void *)plane_array, 0, 0x10000 * 16 * sizeof (int));
00165 while (fgets (instr, MAXSTRING, stdin) != NULL) {
00166     gethex (instr, plane_array, plane); /* read .hex input file and fill plane_array with glyph data */
00167 } /* while not EOF */
00168
00169
00170 /*
00171  Write plane_array glyph data to BMP file as wide or long bitmap.
00172 */
00173 if (wide) {
00174     genwidebmp (plane_array, dpi, tinynum, plane);
00175 }
00176 else {
00177     genlongbmp (plane_array, dpi, tinynum, plane);
00178 }
00179
00180 exit (EXIT_SUCCESS);
00181 }
00182
00183
00184 /**
00185  @brief Output a 4-byte integer in little-endian order.
00186
00187  @param[in] thisword The 4-byte integer to output as binary data.
00188 */
00189 void
00190 output4 (int thisword)
00191 {
00192
00193     putchar (thisword & 0xFF);
00194     putchar ((thisword » 8) & 0xFF);
00195     putchar ((thisword » 16) & 0xFF);
00196     putchar ((thisword » 24) & 0xFF);
00197
00198     return;
00199 }
00200
00201
00202 /**
00203  @brief Output a 2-byte integer in little-endian order.
00204
00205  @param[in] thisword The 2-byte integer to output as binary data.
00206 */
00207 void
00208 output2 (int thisword)
00209 {
00210
00211     putchar (thisword & 0xFF);
00212     putchar ((thisword » 8) & 0xFF);

```

```

00213
00214     return;
00215 }
00216
00217
00218 /**
00219  * @brief Read a Unifont .hex-format input file from stdin.
00220
00221  * Each glyph can be 2, 4, 6, or 8 ASCII hexadecimal digits wide.
00222  * Glyph height is fixed at 16 pixels.
00223
00224  * @param[in] instrstring One line from a Unifont .hex-format file.
00225  * @param[in,out] plane_array Bitmap for this plane, one bitmap row per element.
00226  * @param[in] plane The Unicode plane, 0..17.
00227 */
00228 void
00229 gethex (char *instrstring, int plane_array[0x10000][16], int plane)
00230 {
00231     char *bitstring; /* pointer into instrstring for glyph bitmap */
00232     int i; /* loop variable */
00233     unsigned codept; /* the Unicode code point of the current glyph */
00234     int glyph_plane; /* Unicode plane of current glyph */
00235     int ndigits; /* number of ASCII hexadecimal digits in glyph */
00236     int bytespl; /* bytes per line of pixels in a glyph */
00237     unsigned temprow; /* 1 row of a quadruple-width glyph */
00238     int newrow; /* 1 row of double-width output pixels */
00239     unsigned bitmask; /* to mask off 2 bits of long width glyph */
00240
00241     /*
00242      * Read each input line and place its glyph into the bit array.
00243      */
00244     sscanf (instrstring, "%X", &codept);
00245     glyph_plane = codept >> 16;
00246     if (glyph_plane == plane) {
00247         codept &= 0xFFFF; /* array index will only have 16 bit address */
00248         /* find the colon separator */
00249         for (i = 0; (i < 9) && (instrstring[i] != ':'); i++);
00250         i++; /* position past it */
00251         bitstring = &instrstring[i];
00252         ndigits = strlen (bitstring);
00253         /* don't count '\n' at end of line if present */
00254         if (bitstring[ndigits - 1] == '\n') ndigits--;
00255         bytespl = ndigits >> 5; /* 16 rows per line, 2 digits per byte */
00256
00257         if (bytespl >= 1 && bytespl <= 4) {
00258             for (i = 0; i < 16; i++) { /* 16 rows per glyph */
00259                 /* Read correct number of hexadecimal digits given glyph width */
00260                 switch (bytespl) {
00261                     case 1: sscanf (bitstring, "%2X", &temprow);
00262                             bitstring += 2;
00263                             temprow <= 8; /* left-justify single-width glyph */
00264                             break;
00265                     case 2: sscanf (bitstring, "%4X", &temprow);
00266                             bitstring += 4;
00267                             break;
00268                     /* cases 3 and 4 widths will be compressed by 50% (see below) */
00269                     case 3: sscanf (bitstring, "%6X", &temprow);
00270                             bitstring += 6;
00271                             temprow <= 8; /* left-justify */
00272                             break;
00273                     case 4: sscanf (bitstring, "%8X", &temprow);
00274                             bitstring += 8;
00275                             break;
00276                 } /* switch on number of bytes per row */
00277                 /* compress glyph width by 50% if greater than double-width */
00278                 if (bytespl > 2) {
00279                     newrow = 0x0000;
00280                     /* mask off 2 bits at a time to convert each pair to 1 bit out */
00281                     for (bitmask = 0xC0000000; bitmask != 0; bitmask >>= 2) {
00282                         newrow <= 1;
00283                         if ((temprow & bitmask) != 0) newrow |= 1;
00284                     }
00285                     temprow = newrow;
00286                 } /* done conditioning glyphs beyond double-width */
00287                 plane_array[codept][i] = temprow; /* store glyph bitmap for output */
00288             } /* for each row */
00289         } /* if 1 to 4 bytes per row/line */
00290     } /* if this is the plane we are seeking */
00291
00292     return;
00293 }

```

```

00294
00295
00296 /**
00297  @brief Generate the BMP output file in long format.
00298
00299  This function generates the BMP output file from a bitmap parameter.
00300  This is a long bitmap, 16 glyphs wide by 4,096 glyphs tall.
00301
00302  @param[in] plane_array The array of glyph bitmaps for a plane.
00303  @param[in] dpi Dots per inch, for encoding in the BMP output file header.
00304  @param[in] tinynum Whether to generate tiny numbers in wide grid (unused).
00305  @param[in] plane The Unicode plane, 0..17.
00306 */
00307 void
00308 genlongbmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)
00309 {
00310     char header_string[HDR_LEN]; /* centered header */
00311     char raw_header[HDR_LEN]; /* left-aligned header */
00312     int header[16][16]; /* header row, for chart title */
00313     int hdrlen; /* length of HEADER_STRING */
00314     int startcol; /* column to start printing header, for centering */
00315
00316     unsigned leftcol[0x1000][16]; /* code point legend on left side of chart */
00317     int d1, d2, d3, d4; /* digits for filling leftcol legend */
00318     int codept; /* current starting code point for legend */
00319     int thisrow; /* glyph row currently being rendered */
00320     unsigned toprow[16][16]; /* code point legend on top of chart */
00321     int digitrow; /* row we're in (0..4) for the above hexdigit digits */
00322
00323     /*
00324      DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
00325     */
00326     int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
00327     int ImageSize;
00328     int FileSize;
00329     int Width, Height; /* bitmap image width and height in pixels */
00330     int ppm; /* integer pixels per meter */
00331
00332     int i, j, k;
00333
00334     unsigned bytesout;
00335
00336     void output4(int), output2(int);
00337
00338     /*
00339      Image width and height, in pixels.
00340
00341      N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
00342     */
00343     Width = 18 * 16; /* (2 legend + 16 glyphs) * 16 pixels/glyph */
00344     Height = 4099 * 16; /* (1 header + 4096 glyphs) * 16 rows/glyph */
00345
00346     ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */
00347
00348     FileSize = DataOffset + ImageSize;
00349
00350     /* convert dots/inch to pixels/meter */
00351     if (dpi == 0) dpi = 96;
00352     ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);
00353
00354     /*
00355      Generate the BMP Header
00356     */
00357     putchar ('B');
00358     putchar ('M');
00359
00360     /*
00361      Calculate file size:
00362
00363      BMP Header + InfoHeader + Color Table + Raster Data
00364     */
00365     output4 (FileSize); /* FileSize */
00366     output4 (0x0000); /* reserved */
00367
00368     /* Calculate DataOffset */
00369     output4 (DataOffset);
00370
00371     /*
00372      InfoHeader
00373     */
00374

```

```

00375 output4 (40);          /* Size of InfoHeader */
00376 output4 (Width);       /* Width of bitmap in pixels */
00377 output4 (Height);      /* Height of bitmap in pixels */
00378 output2 (1);           /* Planes (1 plane) */
00379 output2 (1);           /* BitCount (1 = monochrome) */
00380 output4 (0);           /* Compression (0 = none) */
00381 output4 (ImageSize);    /* ImageSize, in bytes */
00382 output4 (ppm);         /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
00383 output4 (ppm);         /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
00384 output4 (2);           /* ColorsUsed (= 2) */
00385 output4 (2);           /* ColorsImportant (= 2) */
00386 output4 (0x00000000); /* black (reserved, B, G, R) */
00387 output4 (0x00FFFFFF); /* white (reserved, B, G, R) */
00388
00389 /*
00390  * Create header row bits.
00391  */
00392 snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
00393 memset ((void *)header, 0, 16 * 16 * sizeof (int)); /* fill with white */
00394 memset ((void *)header_string, ' ', 32 * sizeof (char)); /* 32 spaces */
00395 header_string[32] = '\0'; /* null-terminated */
00396
00397 hdrlen = strlen (raw_header);
00398 if (hdrlen > 32) hdrlen = 32; /* only 32 columns to print header */
00399 startcol = 16 - ((hdrlen + 1) » 1); /* to center header */
00400 /* center up to 32 chars */
00401 memcpy (&header_string[startcol], raw_header, hdrlen);
00402
00403 /* Copy each letter's bitmap from the plane_array[] we constructed. */
00404 /* Each glyph must be single-width, to fit two glyphs in 16 pixels */
00405 for (j = 0; j < 16; j++) {
00406     for (i = 0; i < 16; i++) {
00407         header[i][j] =
00408             (ascii_bits[header_string[j+j ] & 0x7F][i] & 0xFF00) |
00409             (ascii_bits[header_string[j+j+1] & 0x7F][i] » 8);
00410     }
00411 }
00412
00413 /*
00414  * Create the left column legend.
00415  */
00416 memset ((void *)leftcol, 0, 4096 * 16 * sizeof (unsigned));
00417
00418 for (codept = 0x0000; codept < 0x10000; codept += 0x10) {
00419     d1 = (codept » 12) & 0xF; /* most significant hex digit */
00420     d2 = (codept » 8) & 0xF;
00421     d3 = (codept » 4) & 0xF;
00422
00423     thisrow = codept » 4; /* rows of 16 glyphs */
00424
00425     /* fill in first and second digits */
00426     for (digitrow = 0; digitrow < 5; digitrow++) {
00427         leftcol[thisrow][2 + digitrow] =
00428             (hexdigit[d1][digitrow] « 10) |
00429             (hexdigit[d2][digitrow] « 4);
00430     }
00431
00432     /* fill in third digit */
00433     for (digitrow = 0; digitrow < 5; digitrow++) {
00434         leftcol[thisrow][9 + digitrow] = hexdigit[d3][digitrow] « 10;
00435     }
00436     leftcol[thisrow][9 + 4] |= 0xF « 4; /* underscore as 4th digit */
00437
00438     for (i = 0; i < 15; i++) {
00439         leftcol[thisrow][i] |= 0x00000002; /* right border */
00440     }
00441
00442     leftcol[thisrow][15] = 0x0000FFFE; /* bottom border */
00443
00444     if (d3 == 0xF) { /* 256-point boundary */
00445         leftcol[thisrow][15] |= 0x00FF0000; /* longer tic mark */
00446     }
00447
00448     if ((thisrow % 0x40) == 0x3F) { /* 1024-point boundary */
00449         leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
00450     }
00451 }
00452
00453 /*
00454  * Create the top row legend.
00455  */

```

```

00456  memset ((void *)toprow, 0, 16 * 16 * sizeof (unsigned));
00457
00458  for (codept = 0x0; codept <= 0xF; codept++) {
00459      d1 = (codept » 12) & 0xF; /* most significant hex digit */
00460      d2 = (codept » 8) & 0xF;
00461      d3 = (codept » 4) & 0xF;
00462      d4 = codept & 0xF; /* least significant hex digit */
00463
00464      /* fill in last digit */
00465      for (digitrow = 0; digitrow < 5; digitrow++) {
00466          toprow[6 + digitrow][codept] = hexdigit[d4][digitrow] « 6;
00467      }
00468  }
00469
00470  for (j = 0; j < 16; j++) {
00471      /* force bottom pixel row to be white, for separation from glyphs */
00472      toprow[15][j] = 0x0000;
00473  }
00474
00475  /* 1 pixel row with left-hand legend line */
00476  for (j = 0; j < 16; j++) {
00477      toprow[14][j] |= 0xFFFF;
00478  }
00479
00480  /* 14 rows with line on left to fill out this character row */
00481  for (i = 13; i >= 0; i--) {
00482      for (j = 0; j < 16; j++) {
00483          toprow[i][j] |= 0x0001;
00484      }
00485  }
00486
00487  /*
00488   Now write the raster image.
00489
00490   XOR each byte with 0xFF because black = 0, white = 1 in BMP.
00491  */
00492
00493  /* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
00494  for (i = 0xFFFF0; i >= 0; i -= 0x10) {
00495      thisrow = i » 4; /* 16 glyphs per row */
00496      for (j = 15; j >= 0; j--) {
00497          /* left-hand legend */
00498          putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
00499          putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
00500          putchar ((~leftcol[thisrow][j] » 8) & 0xFF);
00501          putchar (~leftcol[thisrow][j] & 0xFF);
00502          /* Unifont glyph */
00503          for (k = 0; k < 16; k++) {
00504              bytesout = ~plane_array[i+k][j] & 0xFFFF;
00505              putchar ((bytesout » 8) & 0xFF);
00506              putchar (~bytesout & 0xFF);
00507          }
00508      }
00509  }
00510
00511  /*
00512   Write the top legend.
00513  */
00514  /* i == 15: bottom pixel row of header is output here */
00515  /* left-hand legend: solid black line except for right-most pixel */
00516  putchar (0x00);
00517  putchar (0x00);
00518  putchar (0x00);
00519  putchar (0x01);
00520  for (j = 0; j < 16; j++) {
00521      putchar ((~toprow[15][j] » 8) & 0xFF);
00522      putchar (~toprow[15][j] & 0xFF);
00523  }
00524
00525  putchar (0xFF);
00526  putchar (0xFF);
00527  putchar (0xFF);
00528  putchar (0xFC);
00529  for (j = 0; j < 16; j++) {
00530      putchar ((~toprow[14][j] » 8) & 0xFF);
00531      putchar (~toprow[14][j] & 0xFF);
00532  }
00533
00534  for (i = 13; i >= 0; i--) {
00535      putchar (0xFF);
00536      putchar (0xFF);

```

```

00537     putchar (0xFF);
00538     putchar (0xFD);
00539     for (j = 0; j < 16; j++) {
00540         putchar ((~toprow[i][j] » 8) & 0xFF);
00541         putchar ( ~toprow[i][j]      & 0xFF);
00542     }
00543 }
00544
00545 /*
00546  * Write the header.
00547  */
00548
00549 /* 7 completely white rows */
00550 for (i = 7; i >= 0; i--) {
00551     for (j = 0; j < 18; j++) {
00552         putchar (0xFF);
00553         putchar (0xFF);
00554     }
00555 }
00556
00557 for (i = 15; i >= 0; i--) {
00558     /* left-hand legend */
00559     putchar (0xFF);
00560     putchar (0xFF);
00561     putchar (0xFF);
00562     putchar (0xFF);
00563     /* header glyph */
00564     for (j = 0; j < 16; j++) {
00565         bytesout = ~header[i][j] & 0xFFFF;
00566         putchar ((bytesout » 8) & 0xFF);
00567         putchar ( bytesout      & 0xFF);
00568     }
00569 }
00570
00571 /* 8 completely white rows at very top */
00572 for (i = 7; i >= 0; i--) {
00573     for (j = 0; j < 18; j++) {
00574         putchar (0xFF);
00575         putchar (0xFF);
00576     }
00577 }
00578
00579 return;
00580 }
00581
00582 /**
00583  * @brief Generate the BMP output file in wide format.
00584  *
00585  * This function generates the BMP output file from a bitmap parameter.
00586  * This is a wide bitmap, 256 glyphs wide by 256 glyphs tall.
00587  *
00588  * @param[in] plane_array The array of glyph bitmaps for a plane.
00589  * @param[in] dpi Dots per inch, for encoding in the BMP output file header.
00590  * @param[in] tinynum Whether to generate tiny numbers in 256x256 grid.
00591  * @param[in] plane The Unicode plane, 0..17.
00592  */
00593 void
00594 genwidebmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)
00595 {
00596     char header_string[257];
00597     char raw_header[HDR_LEN];
00598     int header[16][256]; /* header row, for chart title */
00599     int hstrlen; /* length of HEADER_STRING */
00600     int startcol; /* column to start printing header, for centering */
00601
00602     unsigned leftcol[0x100][16]; /* code point legend on left side of chart */
00603     int d1, d2, d3, d4; /* digits for filling leftcol[][] legend */
00604     int codept; /* current starting code point for legend */
00605     int thisrow; /* glyph row currently being rendered */
00606     unsigned toprw[32][256]; /* code point legend on top of chart */
00607     int digitrow; /* row we're in (0..4) for the above hexdigit digits */
00608     int hexalpha1, hexalpha2; /* to convert hex digits to ASCII */
00609
00610     /*
00611      * DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
00612      */
00613     int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
00614     int ImageSize;
00615     int FileSize;

```



```

00618 int Width, Height; /* bitmap image width and height in pixels */
00619 int ppm; /* integer pixels per meter */
00620
00621 int i, j, k;
00622
00623 unsigned bytesout;
00624
00625 void output4(int), output2(int);
00626
00627 /*
00628  Image width and height, in pixels.
00629
00630  N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
00631 */
00632 Width = 258 * 16; /* ( 2 legend + 256 glyphs) * 16 pixels/glyph */
00633 Height = 260 * 16; /* (2 header + 2 legend + 256 glyphs) * 16 rows/glyph */
00634
00635 ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */
00636
00637 FileSize = DataOffset + ImageSize;
00638
00639 /* convert dots/inch to pixels/meter */
00640 if (dpi == 0) dpi = 96;
00641 ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);
00642
00643 /*
00644  Generate the BMP Header
00645 */
00646 putchar ('B');
00647 putchar ('M');
00648 /*
00649  Calculate file size:
00650
00651  BMP Header + InfoHeader + Color Table + Raster Data
00652 */
00653 output4 (FileSize); /* FileSize */
00654 output4 (0x0000); /* reserved */
00655 /* Calculate DataOffset */
00656 output4 (DataOffset);
00657
00658 /*
00659  InfoHeader
00660 */
00661 output4 (40); /* Size of InfoHeader */
00662 output4 (Width); /* Width of bitmap in pixels */
00663 output4 (Height); /* Height of bitmap in pixels */
00664 output2 (1); /* Planes (1 plane) */
00665 output2 (1); /* BitCount (1 = monochrome) */
00666 output4 (0); /* Compression (0 = none) */
00667 output4 (ImageSize); /* ImageSize, in bytes */
00668 output4 (ppm); /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
00669 output4 (ppm); /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
00670 output4 (2); /* ColorsUsed (= 2) */
00671 output4 (2); /* ColorsImportant (= 2) */
00672 output4 (0x00000000); /* black (reserved, B, G, R) */
00673 output4 (0x00FFFFFF); /* white (reserved, B, G, R) */
00674
00675 /*
00676  Create header row bits.
00677 */
00678 snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
00679 memset ((void *)header, 0, 256 * 16 * sizeof (int)); /* fill with white */
00680 memset ((void *)header_string, ' ', 256 * sizeof (char)); /* 256 spaces */
00681 header_string[256] = '\0'; /* null-terminated */
00682
00683 hdrlen = strlen (raw_header);
00684 /* Wide bitmap can print 256 columns, but limit to 32 columns for long bitmap. */
00685 if (hdrlen > 32) hdrlen = 32;
00686 startcol = 127 - ((hdrlen - 1) » 1); /* to center header */
00687 /* center up to 32 chars */
00688 memcpy (&header_string[startcol], raw_header, hdrlen);
00689
00690 /* Copy each letter's bitmap from the plane_array[] we constructed. */
00691 for (j = 0; j < 256; j++) {
00692     for (i = 0; i < 16; i++) {
00693         header[i][j] = ascii_bits[header_string[j] & 0x7F][i];
00694     }
00695 }
00696
00697 /*
00698  Create the left column legend.

```

```

00699 */
00700 memset ((void *)leftcol, 0, 256 * 16 * sizeof (unsigned));
00701
00702 for (codept = 0x0000; codept < 0x10000; codept += 0x100) {
00703     d1 = (codept » 12) & 0xF; /* most significant hex digit */
00704     d2 = (codept » 8) & 0xF;
00705
00706     thisrow = codept » 8; /* rows of 256 glyphs */
00707
00708     /* fill in first and second digits */
00709
00710     if (tinynum) { /* use 4x5 pixel glyphs */
00711         for (digitrow = 0; digitrow < 5; digitrow++) {
00712             leftcol[thisrow][6 + digitrow] =
00713                 (hexdigit[d1][digitrow] « 10) |
00714                 (hexdigit[d2][digitrow] « 4);
00715         }
00716     }
00717     else { /* bigger numbers -- use glyphs from Unifont itself */
00718         /* convert hexadecimal digits to ASCII equivalent */
00719         hexalpha1 = d1 < 0xA ? '0' + d1 : 'A' + d1 - 0xA;
00720         hexalpha2 = d2 < 0xA ? '0' + d2 : 'A' + d2 - 0xA;
00721
00722         for (i = 0; i < 16; i++) {
00723             leftcol[thisrow][i] =
00724                 (ascii_bits[hexalpha1][i] « 2) |
00725                 (ascii_bits[hexalpha2][i] » 6);
00726         }
00727     }
00728
00729     for (i = 0; i < 15; i++) {
00730         leftcol[thisrow][i] |= 0x00000002; /* right border */
00731     }
00732
00733     leftcol[thisrow][15] = 0x0000FFFE; /* bottom border */
00734
00735     if (d2 == 0xF) { /* 4096-point boundary */
00736         leftcol[thisrow][15] |= 0x00FF0000; /* longer tic mark */
00737     }
00738
00739     if ((thisrow % 0x40) == 0x3F) { /* 16,384-point boundary */
00740         leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
00741     }
00742 }
00743
00744 /*
00745  Create the top row legend.
00746 */
00747 memset ((void *)toprow, 0, 32 * 256 * sizeof (unsigned));
00748
00749 for (codept = 0x00; codept <= 0xFF; codept++) {
00750     d3 = (codept » 4) & 0xF;
00751     d4 = codept & 0xF; /* least significant hex digit */
00752
00753     if (tinynum) {
00754         for (digitrow = 0; digitrow < 5; digitrow++) {
00755             toprow[16 + 6 + digitrow][codept] =
00756                 (hexdigit[d3][digitrow] « 10) |
00757                 (hexdigit[d4][digitrow] « 4);
00758         }
00759     }
00760     else {
00761         /* convert hexadecimal digits to ASCII equivalent */
00762         hexalpha1 = d3 < 0xA ? '0' + d3 : 'A' + d3 - 0xA;
00763         hexalpha2 = d4 < 0xA ? '0' + d4 : 'A' + d4 - 0xA;
00764         for (i = 0; i < 16; i++) {
00765             toprow[14 + i][codept] =
00766                 (ascii_bits[hexalpha1][i] ) |
00767                 (ascii_bits[hexalpha2][i] » 7);
00768         }
00769     }
00770 }
00771
00772 for (j = 0; j < 256; j++) {
00773     /* force bottom pixel row to be white, for separation from glyphs */
00774     toprow[16 + 15][j] = 0x0000;
00775 }
00776
00777 /* 1 pixel row with left-hand legend line */
00778 for (j = 0; j < 256; j++) {
00779     toprow[16 + 14][j] = 0xFFFF;

```

```

00780 }
00781
00782 /* 14 rows with line on left to fill out this character row */
00783 for (i = 13; i >= 0; i--) {
00784     for (j = 0; j < 256; j++) {
00785         toprow[16 + i][j] |= 0x0001;
00786     }
00787 }
00788
00789 /* Form the longer tic marks in top legend */
00790 for (i = 8; i < 16; i++) {
00791     for (j = 0x0F; j < 0x100; j += 0x10) {
00792         toprow[i][j] |= 0x0001;
00793     }
00794 }
00795
00796 /*
00797     Now write the raster image.
00798
00799     XOR each byte with 0xFF because black = 0, white = 1 in BMP.
00800 */
00801
00802 /* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
00803 for (i = 0xFF00; i >= 0; i -= 0x100) {
00804     thisrow = i » 8; /* 256 glyphs per row */
00805     for (j = 15; j >= 0; j--) {
00806         /* left-hand legend */
00807         putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
00808         putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
00809         putchar ((~leftcol[thisrow][j] » 8) & 0xFF);
00810         putchar (~leftcol[thisrow][j] & 0xFF);
00811         /* Unifont glyph */
00812         for (k = 0x00; k < 0x100; k++) {
00813             bytesout = ~plane_array[i+k][j] & 0xFFFF;
00814             putchar ((bytesout » 8) & 0xFF);
00815             putchar (bytesout & 0xFF);
00816         }
00817     }
00818 }
00819
00820 /*
00821     Write the top legend.
00822 */
00823 /* i == 15: bottom pixel row of header is output here */
00824 /* left-hand legend: solid black line except for right-most pixel */
00825 putchar (0x00);
00826 putchar (0x00);
00827 putchar (0x00);
00828 putchar (0x01);
00829 for (j = 0; j < 256; j++) {
00830     putchar ((~toprow[16 + 15][j] » 8) & 0xFF);
00831     putchar (~toprow[16 + 15][j] & 0xFF);
00832 }
00833
00834 putchar (0xFF);
00835 putchar (0xFF);
00836 putchar (0xFF);
00837 putchar (0xFC);
00838 for (j = 0; j < 256; j++) {
00839     putchar ((~toprow[16 + 14][j] » 8) & 0xFF);
00840     putchar (~toprow[16 + 14][j] & 0xFF);
00841 }
00842
00843 for (i = 16 + 13; i >= 0; i--) {
00844     if (i >= 8) { /* make vertical stroke on right */
00845         putchar (0xFF);
00846         putchar (0xFF);
00847         putchar (0xFF);
00848         putchar (0xFD);
00849     }
00850     else { /* all white */
00851         putchar (0xFF);
00852         putchar (0xFF);
00853         putchar (0xFF);
00854         putchar (0xFF);
00855     }
00856     for (j = 0; j < 256; j++) {
00857         putchar ((~toprow[i][j] » 8) & 0xFF);
00858         putchar (~toprow[i][j] & 0xFF);
00859     }
00860 }

```

```

00861
00862  /*
00863   Write the header.
00864  */
00865
00866  /* 8 completely white rows */
00867  for (i = 7; i >= 0; i--) {
00868      for (j = 0; j < 258; j++) {
00869          putchar (0xFF);
00870          putchar (0xFF);
00871      }
00872  }
00873
00874  for (i = 15; i >= 0; i--) {
00875      /* left-hand legend */
00876      putchar (0xFF);
00877      putchar (0xFF);
00878      putchar (0xFF);
00879      putchar (0xFF);
00880      /* header glyph */
00881      for (j = 0; j < 256; j++) {
00882          bytesout = ~header[i][j] & 0xFFFF;
00883          putchar ((bytesout » 8) & 0xFF);
00884          putchar ( bytesout      & 0xFF);
00885      }
00886  }
00887
00888  /* 8 completely white rows at very top */
00889  for (i = 7; i >= 0; i--) {
00890      for (j = 0; j < 258; j++) {
00891          putchar (0xFF);
00892          putchar (0xFF);
00893      }
00894  }
00895
00896  return;
00897 }
00898

```

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 17.0.01"`
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 17.0.01"
```

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 17.0.01" ///< To be printed as chart title.
00033
00034 /**
00035  @brief Array of Unifont ASCII glyphs for chart row & column headings.
00036
00037  Define the array of Unifont ASCII glyphs, code points 0 through 127.
00038  This allows using unifontpic to print charts of glyphs above Unicode
00039  Plane 0. These were copied from font/plane00/unifont-base.hex, plus
00040  U+0020 (ASCII space character).
00041 */
00042 const char *ascii_hex [128] = {
00043  "0000:AAAA00018000000180004A51EA505A51C99E0001800000018000000180005555",
00044  "0001:AAAA00018000000180003993C252325F8A52719380000018000000180005555",
00045  "0002:AAAA00018000000180003BA5C12431198924712580000018000000180005555",
00046  "0003:AAAA00018000000180007BA5C1247919C124792580000018000000180005555",
00047  "0004:AAAA000180000001800079BFC2487A49C248798980000018000000180005555",
00048  "0005:AAAA00018000000180007A4DC2527B53C2D67A4F80000018000000180005555",
00049  "0006:AAAA000180000001800031A5CA287A31CA2849A580000018000000180005555",
00050  "0007:AAAA000180000001800073D1CA1073D1CA1073DF80000018000000180005555",
00051  "0008:AAAA00018000000180001E3991401E3191081E7180000018000000180005555",
00052  "0009:AAAA000180000001800022F9A2203E21A220222180000018000000180005555",
00053  "000A:AAAA000180000001800020F9A08020F9A0803E8180000018000000180005555",
00054  "000B:AAAA000180000001800022F9A22022219420082180000018000000180005555",
00055  "000C:AAAA00018000000180003EF9A0803EF9A080208180000018000000180005555",
00056  "000D:AAAA00018000000180001EF1A08820F1A0901800000018000000180005555",
00057  "000E:AAAA00018000000180001E71A0881C8982883C7180000018000000180005555",
00058  "000F:AAAA00018000000180001EF9A0201C2182203CF980000018000000180005555",
00059  "0010:AAAA0001800000018000391DA510251DA51039DD80000018000000180005555",
00060  "0011:AAAA00018000000180007189CA184A09CA08719D80000018000000180005555",
00061  "0012:AAAA00018000000180007199CA044A09CA10719D80000018000000180005555",
00062  "0013:AAAA00018000000180007199CA044A19CA04719980000018000000180005555",
00063  "0014:AAAA00018000000180007185CA0C4A15CA1C718580000018000000180005555",
00064  "0015:AAAA00018000000180004993EA546A59DBD44A5380000018000000180005555",
00065  "0016:AAAA00018000000180003453C29A31178912711380000018000000180005555",
00066  "0017:AAAA00018000000180007BB9C1247939C124793980000018000000180005555",
00067  "0018:AAAA00018000000180003325C4B447ADC4A43A580000018000000180005555",
00068  "0019:AAAA00018000000180003E89A0D83EA9A0883E8980000018000000180005555",
00069  "001A:AAAA00018000000180003A5DC252325D8A52719D80000018000000180005555",
00070  "001B:AAAA000180000001800079CFC2107991C0507B8F80000018000000180005555",
00071  "001C:AAAA00018000000180001E7190801E61901010E180000018000000180005555",
00072  "001D:AAAA00018000000180000E719080166192100EE180000018000000180005555",
00073  "001E:AAAA00018000000180001C7192801C61941012E180000018000000180005555",
00074  "001F:AAAA000180000001800012719280126192100CE180000018000000180005555",
00075  "0020:00000000000000000000000000000000",
00076  "0021:00000000008080808080808080808000",

```

```
00077 "0022:000022222220000000000000000000",
00078 "0023:000000001212127E24247E4848480000",
00079 "0024:00000000083E4948380E09493E080000",
00080 "0025:00000000314A4A340808162929460000",
00081 "0026:000000001C2222141829454246390000",
00082 "0027:000008080808000000000000000000",
00083 "0028:00000004080810101010101008080400",
00084 "0029:00000020101008080808080810102000",
00085 "002A:00000000000008492A1C2A4908000000",
00086 "002B:0000000000000808087F080808000000",
00087 "002C:0000000000000000000000000018080810",
00088 "002D:00000000000000000000003C0000000000",
00089 "002E:0000000000000000000000000018180000",
00090 "002F:00000000020204080810102040400000",
00091 "0030:00000000182442464A52624224180000",
00092 "0031:000000000818280808080808083E0000",
00093 "0032:000000003C4242020C102040407E0000",
00094 "0033:000000003C4242021C020242423C0000",
00095 "0034:00000000040C142444447E0404040000",
00096 "0035:000000007E4040407C020202423C0000",
00097 "0036:000000001C2040407C424242423C0000",
00098 "0037:000000007E0202040404080808080000",
00099 "0038:000000003C4242423C424242423C0000",
00100 "0039:000000003C4242423E02020204380000",
00101 "003A:00000000000018180000001818000000",
00102 "003B:00000000000018180000001808081000",
00103 "003C:000000000000204081020100804020000",
00104 "003D:000000000000007E0000007E00000000",
00105 "003E:00000000004020100804081020400000",
00106 "003F:000000003C4242020408080008080000",
00107 "0040:000000001C224A565252524E201E0000",
00108 "0041:0000000018242442427E424242420000",
00109 "0042:000000007C4242427C424242427C0000",
00110 "0043:000000003C42424040404042423C0000",
00111 "0044:000000007844424242424244780000",
00112 "0045:000000007E4040407C404040407E0000",
00113 "0046:000000007E4040407C404040400000",
00114 "0047:000000003C424240404E4242463A0000",
00115 "0048:00000000424242427E424242420000",
00116 "0049:000000003E08080808080808083E0000",
00117 "004A:000000001F040404040404444380000",
00118 "004B:0000000042448506060504844420000",
00119 "004C:0000000040404040404040407E0000",
00120 "004D:00000000424266665A5A424242420000",
00121 "004E:0000000042626252524A4A4646420000",
00122 "004F:000000003C42424242424242423C0000",
00123 "0050:000000007C4242427C40404040400000",
00124 "0051:000000003C4242424242425A663C0300",
00125 "0052:000000007C4242427C48444442420000",
00126 "0053:000000003C424240300C0242423C0000",
00127 "0054:000000007F0808080808080808080000",
00128 "0055:000000004242424242424242423C0000",
00129 "0056:0000000041414122222141408080000",
00130 "0057:00000000424242425A5A666642420000",
00131 "0058:00000000424242421818242442420000",
00132 "0059:00000000414122221408080808080000",
00133 "005A:000000007E02020408102040407E0000",
00134 "005B:00000000E0808080808080808080E00",
00135 "005C:00000000404020101008080402020000",
00136 "005D:00000070101010101010101010107000",
00137 "005E:000018244200000000000000000000",
00138 "005F:000000000000000000000000007F00",
00139 "0060:002010080000000000000000000000",
00140 "0061:0000000000003C42023E4242463A0000",
00141 "0062:0000004040405C6242424242625C0000",
00142 "0063:0000000000003C4240404040423C0000",
00143 "0064:0000000202023A4642424242463A0000",
00144 "0065:0000000000003C42427E4040423C0000",
00145 "0066:0000000C1010107C1010101010100000",
00146 "0067:0000000000023A44444438203C42423C",
00147 "0068:0000004040405C624242424242420000",
00148 "0069:000000080800180808080808083E0000",
00149 "006A:0000000404000C0404040404044830",
00150 "006B:00000040404044485060504844420000",
00151 "006C:000000180808080808080808083E0000",
00152 "006D:000000000000764949494949490000",
00153 "006E:0000000000005C6242424242420000",
00154 "006F:0000000000003C4242424242423C0000",
00155 "0070:0000000000005C6242424242625C4040",
00156 "0071:0000000000003A4642424242463A0202",
00157 "0072:0000000000005C6242404040400000",
```



```

00158 "0073:00000000000003C4240300C02423C0000",
00159 "0074:000000001010107C10101010100C0000",
00160 "0075:000000000000424242424242463A0000",
00161 "0076:000000000000424242424242418180000",
00162 "0077:00000000000041494949494949360000",
00163 "0078:000000000000424242418182442420000",
00164 "0079:0000000000004242424242261A02023C",
00165 "007A:0000000000007E0204081020407E0000",
00166 "007B:0000000C10100808102010080810100C",
00167 "007C:00000808080808080808080808080808",
00168 "007D:00000030080810100804081010080830",
00169 "007E:00000031494600000000000000000000",
00170 "007F:AAAA000180000001800073D1CA104BD1CA1073DF800000018000000180005555"
00171 };
00172
00173
00174 /**
00175  @brief Array to hold ASCII bitmaps for chart title.
00176
00177  This array will be created from the strings in ascii_hex[] above.
00178 */
00179 int ascii_bits[128][16];
00180
00181
00182 /**
00183  @brief Array of 4x5 hexadecimal digits for legend.
00184
00185  hexdigit contains 4x5 pixel arrays of tiny digits for the legend.
00186  See unihexgen.c for a more detailed description in the comments.
00187 */
00188 char hexdigit[16][5] = {
00189  {0x6,0x9,0x9,0x9,0x6}, /* 0x0 */
00190  {0x2,0x6,0x2,0x2,0x7}, /* 0x1 */
00191  {0xF,0x1,0xF,0x8,0xF}, /* 0x2 */
00192  {0xE,0x1,0x7,0x1,0xE}, /* 0x3 */
00193  {0x9,0x9,0xF,0x1,0x1}, /* 0x4 */
00194  {0xF,0x8,0xF,0x1,0xF}, /* 0x5 */
00195  {0x6,0x8,0xE,0x9,0x6}, /* 0x6 */
00196  {0xF,0x1,0x2,0x4,0x4}, /* 0x7 */
00197  {0x6,0x9,0x6,0x9,0x6}, /* 0x8 */
00198  {0x6,0x9,0x7,0x1,0x6}, /* 0x9 */
00199  {0xF,0x9,0xF,0x9,0x9}, /* 0xA */
00200  {0xE,0x9,0xE,0x9,0xE}, /* 0xB */
00201  {0x7,0x8,0x8,0x8,0x7}, /* 0xC */
00202  {0xE,0x9,0x9,0x9,0xE}, /* 0xD */
00203  {0xF,0x8,0xE,0x8,0xF}, /* 0xE */
00204  {0xF,0x8,0xE,0x8,0x8}, /* 0xF */
00205 };
00206
00207 #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

Copyright © 2023 Paul Hardy

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 *inf, 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.inf, 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
00133 #ifdef DEBUG
00134             fprintf (params.outf,
00135                     "(U+%04X, U+%04X, U+%04X)\n",
00136                     cho, jung, jong);
00137 #endif
00138             combined_jamo (glyph, cho, jung, jong, tmp_glyph);
00139             print_glyph_hex (params.outf, codept, tmp_glyph);
00140             codept++;
00141             if (jong == JONG_UNICODE_END)
00142                 jong = JONG_EXTB_UNICODE_START - 1; /* Start Extended-B range */
00143         }
00144         if (jung == JUNG_UNICODE_END)
00145             jung = JUNG_EXTB_UNICODE_START - 1; /* Start Extended-B range */
00146     }
00147     if (cho == CHO_UNICODE_END)
00148         cho = CHO_EXTB_UNICODE_START - 1; /* Start Extended-A range */
00149 }
00150
00151 if (params.inf != stdin) fclose (params.inf);
00152 if (params.outf != stdout) fclose (params.outf);
00153
00154 exit (EXIT_SUCCESS);
00155 }

```

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 strcmp (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 (strcmp (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_EXT_A_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 ancient jongseong */
00183         }
00184         /* If starting code point for output Unifont hex file is specified. */
00185         else if (strcmp (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 (strcmp (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                 /*
00199                  * Allow one initial blank glyph at start of a loop, none at end.
00200                  */
00201                 if (params->cho_start < CHO_UNICODE_START) {
00202                     params->cho_start = CHO_UNICODE_START - 1;
00203                 }
00204                 else if (params->cho_start > CHO_UNICODE_END &&
00205                         params->cho_start < CHO_EXT_A_UNICODE_START) {
00206                     params->cho_start = CHO_EXT_A_UNICODE_START - 1;
00207                 }
00208                 /*
00209                  * Do not go past desired Hangul choseong range,
00210                  * Hangul Jamo or Hangul Jamo Extended-A choseong.
00211                  */
00212                 if (params->cho_end > CHO_EXT_A_UNICODE_END) {
00213                     params->cho_end = CHO_EXT_A_UNICODE_END;
00214                 }
00215                 else if (params->cho_end > CHO_UNICODE_END &&
00216                         params->cho_end < CHO_EXT_A_UNICODE_START) {
00217                     params->cho_end = CHO_UNICODE_END;
00218                 }
00219             }
00220             /* If medial vowel (jungseong) range, "jamo 2", get range. */
00221             else if (strcmp (argv [arg_count], "-j2", 3) == 0) {
00222                 arg_count++;
00223                 if (arg_count < argc) {
00224                     get_hex_range (argv [arg_count],
00225                                 &params->jung_start, &params->jung_end);
00226
00227                     /*
00228                      * Allow one initial blank glyph at start of a loop, none at end.
00229                      */
00230                     if (params->jung_start < JUNG_UNICODE_START) {
00231                         params->jung_start = JUNG_UNICODE_START - 1;
00232                     }
00233                     else if (params->jung_start > JUNG_UNICODE_END &&
00234                             params->jung_start < JUNG_EXTB_UNICODE_START) {
00235                         params->jung_start = JUNG_EXTB_UNICODE_START - 1;
00236                     }
00237                     /*
00238                      * Do not go past desired Hangul jungseong range,
00239                      * Hangul Jamo or Hangul Jamo Extended-B jungseong.
00240                      */

```

```

00240     if (params->jung_end > JUNG_EXTB_UNICODE_END) {
00241         params->jung_end = JUNG_EXTB_UNICODE_END;
00242     }
00243     else if (params->jung_end > JUNG_UNICODE_END &&
00244             params->jung_end < JUNG_EXTB_UNICODE_START) {
00245         params->jung_end = JUNG_UNICODE_END;
00246     }
00247 }
00248 }
00249 /* If final consonant (jongseong) range, "jamo 3", get range. */
00250 else if (strcmp (argv [arg_count], "-j3", 3) == 0) {
00251     arg_count++;
00252     if (arg_count < argc) {
00253         get_hex_range (argv [arg_count],
00254                       &params->jong_start, &params->jong_end);
00255         /*
00256          * Allow one initial blank glyph at start of a loop, none at end.
00257          */
00258         if (params->jong_start < JONG_UNICODE_START) {
00259             params->jong_start = JONG_UNICODE_START - 1;
00260         }
00261         else if (params->jong_start > JONG_UNICODE_END &&
00262                 params->jong_start < JONG_EXTB_UNICODE_START) {
00263             params->jong_start = JONG_EXTB_UNICODE_START - 1;
00264         }
00265         /*
00266          * Do not go past desired Hangul jongseong range,
00267          * Hangul Jamo or Hangul Jamo Extended-B jongseong.
00268          */
00269         if (params->jong_end > JONG_EXTB_UNICODE_END) {
00270             params->jong_end = JONG_EXTB_UNICODE_END;
00271         }
00272         else if (params->jong_end > JONG_UNICODE_END &&
00273                 params->jong_end < JONG_EXTB_UNICODE_START) {
00274             params->jong_end = JONG_UNICODE_END;
00275         }
00276     }
00277 }
00278 /* If input file is specified, open it for read access. */
00279 else if (strcmp (argv [arg_count], "-i", 2) == 0) {
00280     arg_count++;
00281     if (arg_count < argc) {
00282         params->infp = fopen (argv [arg_count], "r");
00283         if (params->infp == NULL) {
00284             fprintf (stderr, "\n*** ERROR: Cannot open %s for input.\n\n",
00285                     argv [arg_count]);
00286             exit (EXIT_FAILURE);
00287         }
00288     }
00289 }
00290 /* If output file is specified, open it for write access. */
00291 else if (strcmp (argv [arg_count], "-o", 2) == 0) {
00292     arg_count++;
00293     if (arg_count < argc) {
00294         params->outfp = fopen (argv [arg_count], "w");
00295         if (params->outfp == NULL) {
00296             fprintf (stderr, "\n*** ERROR: Cannot open %s for output.\n\n",
00297                     argv [arg_count]);
00298             exit (EXIT_FAILURE);
00299         }
00300     }
00301 }
00302 /* If help is requested, print help message and exit. */
00303 else if (strcmp (argv [arg_count], "-h", 2) == 0 ||
00304         strcmp (argv [arg_count], "--help", 6) == 0) {
00305     printf ("\nnunigen-hangul [options]\n\n");
00306     printf ("    Generates Hangul syllables from an input Unifont .hex file encoded\n");
00307     printf ("    in Johab 6/3/1 format. By default, the output is the Unicode Hangul\n");
00308     printf ("    Syllables range, U+AC00..U+D7A3. Options allow the user to specify\n");
00309     printf ("    a starting code point for the output Unifont .hex file, and ranges\n");
00310     printf ("    in hexadecimal of the starting and ending Hangul Jamo code points:\n\n");
00311
00312     printf ("        * 1100-115E Initial consonants (choseong)\n");
00313     printf ("        * 1161-11A7 Medial vowels (jungseong)\n");
00314     printf ("        * 11A8-11FF Final consonants (jongseong).\n\n");
00315
00316     printf ("    A single code point or 0 to omit can be specified instead of a range.\n\n");
00317
00318     printf ("    Option    Parameters    Function\n");
00319     printf ("    -----    -\n");
00320     printf ("    -h, --help    Print this message and exit.\n\n");

```

```

00321     printf (" -all          Generate all Hangul syllables, using all modern and\n");
00322     printf ("                ancient Hangul in the Unicode range U+1100..U+11FF,\n");
00323     printf ("                U+A960..U+A97C, and U+D7B0..U+D7FB.\n");
00324     printf ("                WARNING: this will generate over 1,600,000 syllables\n");
00325     printf ("                in a 115 megabyte Unifont .hex format file. The\n");
00326     printf ("                default is to only output modern Hangul syllables.\n\n");
00327     printf (" -c      code_point Starting code point in hexadecimal for output file.\n\n");
00328     printf (" -j1     start-end   Choseong (jamo 1) start-end range in hexadecimal.\n\n");
00329     printf (" -j2     start-end   Jungseong (jamo 2) start-end range in hexadecimal.\n\n");
00330     printf (" -j3     start-end   Jongseong (jamo 3) start-end range in hexadecimal.\n\n");
00331     printf (" -i      input_file  Unifont hangul-base.hex formatted input file.\n\n");
00332     printf (" -o      output_file Unifont .hex format output file.\n\n");
00333     printf (" Example:\n\n");
00334     printf ("     unigen-hangul -c 1 -j3 11AB-11AB -i hangul-base.hex -o nieun-only.hex\n\n");
00335     printf (" Generates Hangul syllables using all modern choseong and jungseong,\n");
00336     printf (" and only the jongseong nieun (Unicode code point U+11AB). The output\n");
00337     printf (" Unifont .hex file will contain code points starting at 1. Instead of\n");
00338     printf (" specifying \"-j3 11AB-11AB\", simply using \"-j3 11AB\" will also suffice.\n\n");
00339
00340     exit (EXIT_SUCCESS);
00341 }
00342
00343     arg_count++;
00344 }
00345
00346 return;
00347 }

```

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.](#)

```

00001 /**
00002  @file unigen-hangul.c
00003
00004  @brief Generate arbitrary hangul syllables.
00005
00006  Input is a Unifont .hex file such as the "hangul-base.hex" file that
00007  is included in the Unifont package.
00008
00009  The default program parameters will generate the Unicode
00010  Hangul Syllables range of U+AC00..U+D7A3. The syllables
00011  will appear in this order:
00012
00013      For each modern choseong {
00014          For each modern jungseong {
00015              Output syllable of choseong and jungseong
00016              For each modern jongseong {
00017                  Output syllable of choseong + jungseong + jongseong
00018              }
00019          }
00020      }
00021
00022  By starting the jongseong code point at one before the first
00023  valid jongseong, the first inner loop iteration will add a
00024  blank glyph for the jongseong portion of the syllable, so
00025  only the current choseong and jungseong will be output first.
00026
00027  @author Paul Hardy
00028
00029  @copyright Copyright © 2023 Paul Hardy
00030 */
00031 /*
00032  LICENSE:
00033
00034  This program is free software: you can redistribute it and/or modify
00035  it under the terms of the GNU General Public License as published by
00036  the Free Software Foundation, either version 2 of the License, or
00037  (at your option) any later version.
00038
00039  This program is distributed in the hope that it will be useful,
00040  but WITHOUT ANY WARRANTY; without even the implied warranty of
00041  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

```

```

00042     GNU General Public License for more details.
00043
00044     You should have received a copy of the GNU General Public License
00045     along with this program.  If not, see <http://www.gnu.org/licenses/>.
00046 */
00047
00048 #include <stdio.h>
00049 #include <stdlib.h>
00050 #include "hangul.h"
00051
00052 // #define DEBUG
00053
00054
00055 struct PARAMS {
00056     unsigned starting_codept; /* First output Unicode code point. */
00057     unsigned cho_start, cho_end; /* Choseong start and end code points. */
00058     unsigned jung_start, jung_end; /* Jungseong start and end code points. */
00059     unsigned jong_start, jong_end; /* Jongseong start and end code points. */
00060     FILE *infp;
00061     FILE *outfp;
00062 };
00063
00064
00065 /**
00066  @brief Program entry point.
00067 */
00068 int
00069 main (int argc, char *argv[]) {
00070     int i; /* loop variable */
00071     unsigned codept;
00072     unsigned max_codept;
00073     unsigned glyph[MAX_GLYPHS][16];
00074     unsigned tmp_glyph [16]; /* To build one combined glyph at a time. */
00075     int cho, jung, jong; /* The 3 components in a Hangul syllable. */
00076
00077     /// Default parameters for Hangul syllable generation.
00078     struct PARAMS params = { 0xAC00, /* Starting output Unicode code point */
00079         0x1100, /* First modern choseong */
00080         0x1112, /* Last modern choseong */
00081         0x1161, /* First modern jungseong */
00082         0x1175, /* Last modern jungseong */
00083         0x11A7, /* One before first modern jongseong */
00084         0x11C2, /* Last modern jongseong */
00085         stdin, /* Default input file pointer */
00086         stdout /* Default output file pointer */
00087     };
00088
00089     void parse_args (int argc, char *argv[], struct PARAMS *params);
00090
00091     unsigned hangul_read_base16 (FILE *infp, unsigned glyph[][16]);
00092
00093     void print_glyph_hex (FILE *fp, unsigned codept, unsigned *this_glyph);
00094
00095     void combined_jamo (unsigned glyph [MAX_GLYPHS][16],
00096         unsigned cho, unsigned jung, unsigned jong,
00097         unsigned *combined_glyph);
00098
00099     if (argc > 1) {
00100         parse_args (argc, argv, &params);
00101     }
00102     #ifndef DEBUG
00103         fprintf (stderr,
00104             "Range: (U+%04X, U+%04X, U+%04X) to (U+%04X, U+%04X, U+%04X)\n",
00105             params.cho_start, params.jung_start, params.jong_start,
00106             params.cho_end, params.jung_end, params.jong_end);
00107     #endif
00108 }
00109
00110 /*
00111  Initialize glyph array to all zeroes.
00112 */
00113 for (codept = 0; codept < MAX_GLYPHS; codept++) {
00114     for (i = 0; i < 16; i++) glyph[codept][i] = 0x0000;
00115 }
00116
00117 /*
00118  Read Hangul base glyph file.
00119 */
00120 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
00133 #ifdef DEBUG
00134             fprintf (params.outfp,
00135                 "(U+%04X, U+%04X, U+%04X)\n",
00136                 cho, jung, jong);
00137 #endif
00138             combined_jamo (glyph, cho, jung, jong, tmp_glyph);
00139             print_glyph_hex (params.outfp, codept, tmp_glyph);
00140             codept++;
00141             if (jong == JONG_UNICODE_END)
00142                 jong = JONG_EXTB_UNICODE_START - 1; /* Start Extended-B range */
00143         }
00144         if (jung == JUNG_UNICODE_END)
00145             jung = JUNG_EXTB_UNICODE_START - 1; /* Start Extended-B range */
00146     }
00147     if (cho == CHO_UNICODE_END)
00148         cho = CHO_EXTB_UNICODE_START - 1; /* Start Extended-A range */
00149 }
00150
00151 if (params.infp != stdin) fclose (params.infp);
00152 if (params.outfp != stdout) fclose (params.outfp);
00153
00154 exit (EXIT_SUCCESS);
00155 }
00156
00157 /**
00158  * @brief Parse command line arguments.
00159  */
00160 void
00161 parse_args (int argc, char *argv[], struct PARAMS *params) {
00162     int arg_count; /* Current index into argv. */
00163
00164     void get_hex_range (char *instring, unsigned *start, unsigned *end);
00165
00166     int strncmp (const char *s1, const char *s2, size_t n);
00167
00168     arg_count = 1;
00169
00170     while (arg_count < argc) {
00171         /* If all 600,000+ Hangul syllables are requested. */
00172         if (strncmp (argv [arg_count], "-all", 4) == 0) {
00173             params->starting_codept = 0x0001;
00174             params->cho_start = CHO_UNICODE_START; /* First modern choseong */
00175             params->cho_end = CHO_EXTB_UNICODE_END; /* Last ancient choseong */
00176             params->jung_start = JUNG_UNICODE_START; /* First modern jungseong */
00177             params->jung_end = JUNG_EXTB_UNICODE_END; /* Last ancient jungseong */
00178             params->jong_start = JONG_UNICODE_START - 1; /* One before first modern jongseong */
00179             params->jong_end = JONG_EXTB_UNICODE_END; /* Last ancient jongseong */
00180         }
00181         /* If starting code point for output Unifont hex file is specified. */
00182         else if (strncmp (argv [arg_count], "-c", 2) == 0) {
00183             arg_count++;
00184             if (arg_count < argc) {
00185                 sscanf (argv [arg_count], "%X", &params->starting_codept);
00186             }
00187         }
00188         /* If initial consonant (choseong) range, "jamo 1", get range. */
00189         else if (strncmp (argv [arg_count], "-j1", 3) == 0) {
00190             arg_count++;
00191             if (arg_count < argc) {
00192                 get_hex_range (argv [arg_count],
00193                     &params->cho_start, &params->cho_end);
00194             }
00195             /*
00196              * Allow one initial blank glyph at start of a loop, none at end.
00197              */
00198             if (params->cho_start < CHO_UNICODE_START) {
00199                 params->cho_start = CHO_UNICODE_START - 1;
00200             }
00201             else if (params->cho_start > CHO_UNICODE_END &&

```

```

00204         params->cho_start < CHO_EXTA_UNICODE_START) {
00205     params->cho_start = CHO_EXTA_UNICODE_START - 1;
00206 }
00207 /*
00208     Do not go past desired Hangul choseong range,
00209     Hangul Jamo or Hangul Jamo Extended-A choseong.
00210 */
00211 if (params->cho_end > CHO_EXTA_UNICODE_END) {
00212     params->cho_end = CHO_EXTA_UNICODE_END;
00213 }
00214 else if (params->cho_end > CHO_UNICODE_END &&
00215         params->cho_end < CHO_EXTA_UNICODE_START) {
00216     params->cho_end = CHO_UNICODE_END;
00217 }
00218 }
00219 }
00220 /* If medial vowel (jungseong) range, "jamo 2", get range. */
00221 else if (strcmp (argv [arg_count], "-j2", 3) == 0) {
00222     arg_count++;
00223     if (arg_count < argc) {
00224         get_hex_range (argv [arg_count],
00225             &params->jung_start, &params->jung_end);
00226     /*
00227         Allow one initial blank glyph at start of a loop, none at end.
00228     */
00229     if (params->jung_start < JUNG_UNICODE_START) {
00230         params->jung_start = JUNG_UNICODE_START - 1;
00231     }
00232     else if (params->jung_start > JUNG_UNICODE_END &&
00233             params->jung_start < JUNG_EXTB_UNICODE_START) {
00234         params->jung_start = JUNG_EXTB_UNICODE_START - 1;
00235     }
00236     /*
00237         Do not go past desired Hangul jungseong range,
00238         Hangul Jamo or Hangul Jamo Extended-B jungseong.
00239     */
00240     if (params->jung_end > JUNG_EXTB_UNICODE_END) {
00241         params->jung_end = JUNG_EXTB_UNICODE_END;
00242     }
00243     else if (params->jung_end > JUNG_UNICODE_END &&
00244             params->jung_end < JUNG_EXTB_UNICODE_START) {
00245         params->jung_end = JUNG_UNICODE_END;
00246     }
00247 }
00248 }
00249 /* If final consonant (jongseong) range, "jamo 3", get range. */
00250 else if (strcmp (argv [arg_count], "-j3", 3) == 0) {
00251     arg_count++;
00252     if (arg_count < argc) {
00253         get_hex_range (argv [arg_count],
00254             &params->jong_start, &params->jong_end);
00255     /*
00256         Allow one initial blank glyph at start of a loop, none at end.
00257     */
00258     if (params->jong_start < JONG_UNICODE_START) {
00259         params->jong_start = JONG_UNICODE_START - 1;
00260     }
00261     else if (params->jong_start > JONG_UNICODE_END &&
00262             params->jong_start < JONG_EXTB_UNICODE_START) {
00263         params->jong_start = JONG_EXTB_UNICODE_START - 1;
00264     }
00265     /*
00266         Do not go past desired Hangul jongseong range,
00267         Hangul Jamo or Hangul Jamo Extended-B jongseong.
00268     */
00269     if (params->jong_end > JONG_EXTB_UNICODE_END) {
00270         params->jong_end = JONG_EXTB_UNICODE_END;
00271     }
00272     else if (params->jong_end > JONG_UNICODE_END &&
00273             params->jong_end < JONG_EXTB_UNICODE_START) {
00274         params->jong_end = JONG_UNICODE_END;
00275     }
00276 }
00277 }
00278 /* If input file is specified, open it for read access. */
00279 else if (strcmp (argv [arg_count], "-i", 2) == 0) {
00280     arg_count++;
00281     if (arg_count < argc) {
00282         params->infp = fopen (argv [arg_count], "r");
00283         if (params->infp == NULL) {
00284             fprintf (stderr, "\n*** ERROR: Cannot open %s for input.\n\n",

```

```

00285         argv[arg_count]);
00286     exit (EXIT_FAILURE);
00287 }
00288 }
00289 }
00290 /* If output file is specified, open it for write access. */
00291 else if (strcmp (argv [arg_count], "-o", 2) == 0) {
00292     arg_count++;
00293     if (arg_count < argc) {
00294         params->outfp = fopen (argv [arg_count], "w");
00295         if (params->outfp == NULL) {
00296             fprintf (stderr, "\n*** ERROR: Cannot open %s for output.\n\n",
00297                     argv [arg_count]);
00298             exit (EXIT_FAILURE);
00299         }
00300     }
00301 }
00302 /* If help is requested, print help message and exit. */
00303 else if (strcmp (argv [arg_count], "-h", 2) == 0 ||
00304          strcmp (argv [arg_count], "--help", 6) == 0) {
00305     printf ("\nunigen-hangul [options]\n\n");
00306     printf ("    Generates Hangul syllables from an input Unifont .hex file encoded\n");
00307     printf ("    in Johab 6/3/1 format. By default, the output is the Unicode Hangul\n");
00308     printf ("    Syllables range, U+AC00..U+D7A3. Options allow the user to specify\n");
00309     printf ("    a starting code point for the output Unifont .hex file, and ranges\n");
00310     printf ("    in hexadecimal of the starting and ending Hangul Jamo code points:\n\n");
00311
00312     printf ("        * 1100-115E Initial consonants (choseong)\n");
00313     printf ("        * 1161-11A7 Medial vowels (jungseong)\n");
00314     printf ("        * 11A8-11FF Final consonants (jongseong).\n\n");
00315
00316     printf ("    A single code point or 0 to omit can be specified instead of a range.\n\n");
00317
00318     printf ("    Option   Parameters   Function\n");
00319     printf ("    -----   -\n");
00320     printf ("    -h, --help       Print this message and exit.\n\n");
00321     printf ("    -all              Generate all Hangul syllables, using all modern and\n");
00322     printf ("                    ancient Hangul in the Unicode range U+1100..U+11FF,\n");
00323     printf ("                    U+A960..U+A97C, and U+D7B0..U+D7FB.\n");
00324     printf ("                    WARNING: this will generate over 1,600,000 syllables\n");
00325     printf ("                    in a 115 megabyte Unifont .hex format file. The\n");
00326     printf ("                    default is to only output modern Hangul syllables.\n\n");
00327     printf ("    -c      code_point  Starting code point in hexadecimal for output file.\n\n");
00328     printf ("    -j1     start-end   Choseong (jamo 1) start-end range in hexadecimal.\n\n");
00329     printf ("    -j2     start-end   Jungseong (jamo 2) start-end range in hexadecimal.\n\n");
00330     printf ("    -j3     start-end   Jongseong (jamo 3) start-end range in hexadecimal.\n\n");
00331     printf ("    -i      input_file  Unifont hangul-base.hex formatted input file.\n\n");
00332     printf ("    -o      output_file Unifont .hex format output file.\n\n");
00333     printf ("    Example:\n\n");
00334     printf ("    unigen-hangul -c 1 -j3 11AB-11AB -i hangul-base.hex -o nieun-only.hex\n\n");
00335     printf ("    Generates Hangul syllables using all modern choseong and jungseong,\n");
00336     printf ("    and only the jongseong nieun (Unicode code point U+11AB). The output\n");
00337     printf ("    Unifont .hex file will contain code points starting at 1. Instead of\n");
00338     printf ("    specifying \"-j3 11AB-11AB\", simply using \"-j3 11AB\" will also suffice.\n\n");
00339     exit (EXIT_SUCCESS);
00340 }
00341 }
00342
00343     arg_count++;
00344 }
00345
00346 return;
00347 }
00348
00349 /**
00350  * @brief Scan a hexadecimal range from a character string.
00351  */
00352 void
00353 get_hex_range (char *instring, unsigned *start, unsigned *end) {
00354     int i; /* String index variable. */
00355
00356     /* Get first number in range. */
00357     sscanf (instring, "%X", start);
00358     for (i = 0;
00359          instring[i] != '\0' && instring[i] != '-';
00360          i++);
00361     /* Get last number in range. */
00362     if (instring[i] == '-') {
00363         i++;
00364     }
00365     sscanf (instring + i, "%X", end);
00366 }

```

```
00366     sscanf (&instring [i], "%X", end);
00367 }
00368 else {
00369     *end = *start;
00370 }
00371
00372 return;
00373 }
```

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

Copyright (C) 2013, Paul Hardy.

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 66 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 225 of file [unigencircles.c](#).

```
00226 {
00227
00228     char newstring[256];
00229     /* Circle hex string pattern is "00000008000024004200240000000000" */
00230
00231     /* For double diacritical glyphs (offset = -8) */
00232     /* Combining circle is left-justified. */
00233     char circle08[64]={0x0,0x0,0x0,0x0, /* row 1 */
00234                       0x0,0x0,0x0,0x0, /* row 2 */
00235                       0x0,0x0,0x0,0x0, /* row 3 */
00236                       0x0,0x0,0x0,0x0, /* row 4 */
00237                       0x0,0x0,0x0,0x0, /* row 5 */
00238                       0x0,0x0,0x0,0x0, /* row 6 */
00239                       0x2,0x4,0x0,0x0, /* row 7 */
00240                       0x0,0x0,0x0,0x0, /* row 8 */
00241                       0x4,0x2,0x0,0x0, /* row 9 */
00242                       0x0,0x0,0x0,0x0, /* row 10 */
00243                       0x2,0x4,0x0,0x0, /* row 11 */
00244                       0x0,0x0,0x0,0x0, /* row 12 */
00245                       0x0,0x0,0x0,0x0, /* row 13 */
00246                       0x0,0x0,0x0,0x0, /* row 14 */
00247                       0x0,0x0,0x0,0x0, /* row 15 */
00248                       0x0,0x0,0x0,0x0}; /* row 16 */
00249
00250     /* For all other combining glyphs (offset = -16) */
00251     /* Combining circle is centered in 16 columns. */
```

```

00252 char circle16[64]={0x0,0x0,0x0,0x0, /* row 1 */
00253                    0x0,0x0,0x0,0x0, /* row 2 */
00254                    0x0,0x0,0x0,0x0, /* row 3 */
00255                    0x0,0x0,0x0,0x0, /* row 4 */
00256                    0x0,0x0,0x0,0x0, /* row 5 */
00257                    0x0,0x0,0x0,0x0, /* row 6 */
00258                    0x0,0x2,0x4,0x0, /* row 7 */
00259                    0x0,0x0,0x0,0x0, /* row 8 */
00260                    0x0,0x4,0x2,0x0, /* row 9 */
00261                    0x0,0x0,0x0,0x0, /* row 10 */
00262                    0x0,0x2,0x4,0x0, /* row 11 */
00263                    0x0,0x0,0x0,0x0, /* row 12 */
00264                    0x0,0x0,0x0,0x0, /* row 13 */
00265                    0x0,0x0,0x0,0x0, /* row 14 */
00266                    0x0,0x0,0x0,0x0, /* row 15 */
00267                    0x0,0x0,0x0,0x0}; /* row 16 */
00268
00269 char *circle; /* points into circle16 or circle08 */
00270
00271 int digit1, digit2; /* corresponding digits in each string */
00272
00273 int i; /* index variables */
00274
00275
00276 /*
00277  Determine if combining circle is left-justified (offset = -8)
00278  or centered (offset = -16).
00279 */
00280 circle = (offset >= -8) ? circle08 : circle16;
00281
00282 /* for each character position, OR the corresponding circle glyph value */
00283 for (i = 0; i < 64; i++) {
00284     glyphstring[i] = toupper (glyphstring[i]);
00285
00286     /* Convert ASCII character to a hexadecimal integer */
00287     digit1 = (glyphstring[i] <= '9') ?
00288             (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);
00289
00290     /* Superimpose dashed circle */
00291     digit2 = digit1 | circle[i];
00292
00293     /* Convert hexadecimal integer to an ASCII character */
00294     newstring[i] = (digit2 <= 9) ?
00295             ('0' + digit2) : ('A' + digit2 - 0xA);
00296 }
00297
00298 /* Terminate string for output */
00299 newstring[i++] = '\n';
00300 newstring[i++] = '\0';
00301
00302 memcpy (glyphstring, newstring, i);
00303
00304 return;
00305 }

```

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 167 of file [unigencircles.c](#).

```

00168 {
00169
00170     char newstring[256];
00171     /* Circle hex string pattern is "00000008000024004200240000000000" */
00172     char circle[32]={0x0,0x0, /* row 1 */
00173                     0x0,0x0, /* row 2 */
00174                     0x0,0x0, /* row 3 */
00175                     0x0,0x0, /* row 4 */
00176                     0x0,0x0, /* row 5 */
00177                     0x0,0x0, /* row 6 */
00178                     0x2,0x4, /* row 7 */
00179                     0x0,0x0, /* row 8 */
00180                     0x4,0x2, /* row 9 */
00181                     0x0,0x0, /* row 10 */
00182                     0x2,0x4, /* row 11 */
00183                     0x0,0x0, /* row 12 */
00184                     0x0,0x0, /* row 13 */
00185                     0x0,0x0, /* row 14 */
00186                     0x0,0x0, /* row 15 */
00187                     0x0,0x0}; /* row 16 */
00188
00189     int digit1, digit2; /* corresponding digits in each string */
00190
00191     int i; /* index variables */
00192
00193     /* for each character position, OR the corresponding circle glyph value */
00194     for (i = 0; i < 32; i++) {
00195         glyphstring[i] = toupper (glyphstring[i]);
00196
00197         /* Convert ASCII character to a hexadecimal integer */
00198         digit1 = (glyphstring[i] <= '9') ?
00199             (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);
00200
00201         /* Superimpose dashed circle */
00202         digit2 = digit1 | circle[i];
00203
00204         /* Convert hexadecimal integer to an ASCII character */
00205         newstring[i] = (digit2 <= 9) ?
00206             ('0' + digit2) : ('A' + digit2 - 0xA);
00207     }
00208
00209     /* Terminate string for output */
00210     newstring[i++] = '\n';
00211     newstring[i++] = '\0';
00212
00213     memcpy (glyphstring, newstring, i);
00214
00215     return;
00216 }

```

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 77 of file `unigencircles.c`.

```

00078 {
00079
00080     char teststring[MAXSTRING]; /* current input line */
00081     unsigned loc; /* Unicode code point of current input line */
00082     int offset; /* offset value of a combining character */
00083     char *gstart; /* glyph start, pointing into teststring */
00084
00085     char combining[0x110000]; /* 1 --> combining glyph; 0 --> non-combining */
00086     char x_offset [0x110000]; /* second value in *combining.txt files */
00087
00088     void add_single_circle(char *); /* add a single-width dashed circle */
00089     void add_double_circle(char *, int); /* add a double-width dashed circle */
00090
00091     FILE *infilefp;
00092
00093     /*
00094      * if (argc != 3) {
00095      *     fprintf (stderr,
00096      *         "\n\nUsage: %s combining.txt nonprinting.hex < unifont.hex > unifontfull.hex\n\n");
00097      *     exit (EXIT_FAILURE);
00098      * }
00099     */
00100
00101     /*
00102      * Read the combining characters list.
00103      */
00104     /* Start with no combining code points flagged */
00105     memset (combining, 0, 0x110000 * sizeof (char));
00106     memset (x_offset , 0, 0x110000 * sizeof (char));
00107
00108     if ((infilefp = fopen (argv[1], "r")) == NULL) {
00109         fprintf (stderr, "ERROR - combining characters file %s not found.\n\n",
00110             argv[1]);
00111         exit (EXIT_FAILURE);
00112     }
00113
00114     /* Flag list of combining characters to add a dashed circle. */
00115     while (fscanf (infilefp, "%X:%d", &loc, &offset) != EOF) {
00116         /*
00117          * U+01107F and U+01D1A0 are not defined as combining characters
00118          * in Unicode; they were added in a combining.txt file as the
00119          * only way to make them look acceptable in proximity to other
00120          * glyphs in their script.
00121          */
00122         if (loc != 0x01107F && loc != 0x01D1A0) {
00123             combining[loc] = 1;
00124             x_offset [loc] = offset;
00125         }
00126     }
00127     fclose (infilefp); /* all done reading combining.txt */
00128
00129     /* Now read the non-printing glyphs; they never have dashed circles */
00130     if ((infilefp = fopen (argv[2], "r")) == NULL) {
00131         fprintf (stderr, "ERROR - nonprinting characters file %s not found.\n\n",
00132             argv[1]);
00133         exit (EXIT_FAILURE);
00134     }
00135
00136     /* Reset list of nonprinting characters to avoid adding a dashed circle. */
00137     while (fscanf (infilefp, "%X:%s", &loc) != EOF) combining[loc] = 0;
00138
00139     fclose (infilefp); /* all done reading nonprinting.hex */
00140
00141     /*
00142      * Read the hex glyphs.
00143      */
00144     teststring[MAXSTRING - 1] = '\0'; /* so there's no chance we leave array */
00145     while (fgets (teststring, MAXSTRING-1, stdin) != NULL) {
00146         sscanf (teststring, "%X", &loc); /* loc == the Unicode code point */
00147         gstart = strchr (teststring, ':') + 1; /* start of glyph bitmap */
00148         if (combining[loc]) { /* if a combining character */
00149             if (strlen (gstart) < 35)
00150                 add_single_circle (gstart); /* single-width */
00151             else

```



```

00152     add_double_circle (gstart, x_offset[loc]); /* double-width */
00153 }
00154 printf ("%s", teststring); /* output the new character .hex string */
00155 }
00156
00157 exit (EXIT_SUCCESS);
00158 }

```

Here is the call graph for this function:

5.30 unigencircles.c

[Go to the documentation of this file.](#)

```

00001 /**
00002  @file unigencircles.c
00003
00004  @brief unigencircles - Superimpose dashed combining circles
00005         on combining glyphs
00006
00007  @author Paul Hardy
00008
00009  @copyright Copyright (C) 2013, Paul Hardy.
00010 */
00011 /*
00012  LICENSE:
00013
00014  This program is free software: you can redistribute it and/or modify
00015  it under the terms of the GNU General Public License as published by
00016  the Free Software Foundation, either version 2 of the License, or
00017  (at your option) any later version.
00018
00019  This program is distributed in the hope that it will be useful,
00020  but WITHOUT ANY WARRANTY; without even the implied warranty of
00021  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00022  GNU General Public License for more details.
00023
00024  You should have received a copy of the GNU General Public License
00025  along with this program. If not, see <http://www.gnu.org/licenses/>.
00026 */
00027
00028 /*
00029  8 July 2017 [Paul Hardy]:
00030  - Reads new second field that contains an x-axis offset for
00031    each combining character in "combining.txt" files.
00032  - Uses the above x-axis offset value for a combining character
00033    to print combining circle in the left half of a double
00034    diacritic combining character grid, or in the center for
00035    other combining characters.
00036  - Adds exceptions for U+01107F (Brahmi number joiner) and
00037    U+01D1A0 (vertical stroke musical ornament); they are in
00038    a combining.txt file for positioning, but are not actually
00039    Unicode combining characters.
00040  - Typo fix: "single-width"-->"double-width" in comment for
00041    add_double_circle function.
00042
00043  12 August 2017 [Paul Hardy]:
00044  - Hard-code Miao vowels to show combining circles after
00045    removing them from font/plane01/plane01-combining.txt.
00046
00047  26 December 2017 [Paul Hardy]:
00048  - Remove Miao hard-coding; they are back in unibmp2hex.c and
00049    in font/plane01/plane01-combining.txt.
00050
00051  11 May 2019 [Paul Hardy]:
00052  - Changed strncpy calls to memcpy calls to avoid a compiler
00053    warning.
00054
00055  6 September 2025 [Paul Hardy]:
00056  - Changed loc from "int" to "unsigned" for compatibility with
00057    fscanf and sscanf definitions.
00058 */
00059
00060
00061 #include <stdio.h>

```

```

00062 #include <stdlib.h>
00063 #include <string.h>
00064 #include <ctype.h>
00065
00066 #define MAXSTRING 256 ///< Maximum input line length - 1.
00067
00068
00069 /**
00070  @brief The main function.
00071
00072  @param[in] argc The count of command line arguments.
00073  @param[in] argv Pointer to array of command line arguments.
00074  @return This program exits with status EXIT_SUCCESS.
00075 */
00076 int
00077 main (int argc, char **argv)
00078 {
00079
00080     char teststring[MAXSTRING]; /* current input line */
00081     unsigned loc; /* Unicode code point of current input line */
00082     int offset; /* offset value of a combining character */
00083     char *gstart; /* glyph start, pointing into teststring */
00084
00085     char combining[0x110000]; /* 1 --> combining glyph; 0 --> non-combining */
00086     char x_offset [0x110000]; /* second value in *combining.txt files */
00087
00088     void add_single_circle(char *); /* add a single-width dashed circle */
00089     void add_double_circle(char *, int); /* add a double-width dashed circle */
00090
00091     FILE *infilefp;
00092
00093     /*
00094      if (argc != 3) {
00095          fprintf (stderr,
00096                  "\n\nUsage: %s combining.txt nonprinting.hex < unifont.hex > unifontfull.hex\n\n");
00097          exit (EXIT_FAILURE);
00098      }
00099     */
00100
00101     /*
00102      Read the combining characters list.
00103     */
00104     /* Start with no combining code points flagged */
00105     memset (combining, 0, 0x110000 * sizeof (char));
00106     memset (x_offset , 0, 0x110000 * sizeof (char));
00107
00108     if ((infilefp = fopen (argv[1], "r")) == NULL) {
00109         fprintf (stderr, "ERROR - combining characters file %s not found.\n\n",
00110                 argv[1]);
00111         exit (EXIT_FAILURE);
00112     }
00113
00114     /* Flag list of combining characters to add a dashed circle. */
00115     while (fscanf (infilefp, "%X:%d", &loc, &offset) != EOF) {
00116         /*
00117          U+01107F and U+01D1A0 are not defined as combining characters
00118          in Unicode; they were added in a combining.txt file as the
00119          only way to make them look acceptable in proximity to other
00120          glyphs in their script.
00121         */
00122         if (loc != 0x01107F && loc != 0x01D1A0) {
00123             combining[loc] = 1;
00124             x_offset [loc] = offset;
00125         }
00126     }
00127     fclose (infilefp); /* all done reading combining.txt */
00128
00129     /* Now read the non-printing glyphs; they never have dashed circles */
00130     if ((infilefp = fopen (argv[2], "r")) == NULL) {
00131         fprintf (stderr, "ERROR - nonprinting characters file %s not found.\n\n",
00132                 argv[1]);
00133         exit (EXIT_FAILURE);
00134     }
00135
00136     /* Reset list of nonprinting characters to avoid adding a dashed circle. */
00137     while (fscanf (infilefp, "%X:%s", &loc) != EOF) combining[loc] = 0;
00138
00139     fclose (infilefp); /* all done reading nonprinting.hex */
00140
00141     /*
00142      Read the hex glyphs.

```

```

00143 */
00144 teststring[MAXSTRING - 1] = '\0'; /* so there's no chance we leave array */
00145 while (fgets (teststring, MAXSTRING-1, stdin) != NULL) {
00146     sscanf (teststring, "%X", &loc); /* loc == the Unioecode code point */
00147     gstart = strchr (teststring, ':') + 1; /* start of glyph bitmap */
00148     if (combining[loc]) { /* if a combining character */
00149         if (strlen (gstart) < 35)
00150             add_single_circle (gstart); /* single-width */
00151         else
00152             add_double_circle (gstart, x_offset[loc]); /* double-width */
00153     }
00154     printf ("%s", teststring); /* output the new character .hex string */
00155 }
00156
00157 exit (EXIT_SUCCESS);
00158 }
00159
00160
00161 /**
00162  @brief Superimpose a single-width dashed combining circle on a glyph bitmap.
00163
00164  @param[in,out] glyphstring A single-width glyph, 8x16 pixels.
00165 */
00166 void
00167 add_single_circle (char *glyphstring)
00168 {
00169     char newstring[256];
00170     /* Circle hex string pattern is "00000008000024004200240000000000" */
00171     char circle[32]={0x0,0x0, /* row 1 */
00172                     0x0,0x0, /* row 2 */
00173                     0x0,0x0, /* row 3 */
00174                     0x0,0x0, /* row 4 */
00175                     0x0,0x0, /* row 5 */
00176                     0x0,0x0, /* row 6 */
00177                     0x2,0x4, /* row 7 */
00178                     0x0,0x0, /* row 8 */
00179                     0x4,0x2, /* row 9 */
00180                     0x0,0x0, /* row 10 */
00181                     0x2,0x4, /* row 11 */
00182                     0x0,0x0, /* row 12 */
00183                     0x0,0x0, /* row 13 */
00184                     0x0,0x0, /* row 14 */
00185                     0x0,0x0, /* row 15 */
00186                     0x0,0x0}; /* row 16 */
00187
00188     int digit1, digit2; /* corresponding digits in each string */
00189
00190     int i; /* index variables */
00191
00192     /* for each character position, OR the corresponding circle glyph value */
00193     for (i = 0; i < 32; i++) {
00194         glyphstring[i] = toupper (glyphstring[i]);
00195
00196         /* Convert ASCII character to a hexadecimal integer */
00197         digit1 = (glyphstring[i] <= '9') ?
00198             (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);
00199
00200         /* Superimpose dashed circle */
00201         digit2 = digit1 | circle[i];
00202
00203         /* Convert hexadecimal integer to an ASCII character */
00204         newstring[i] = (digit2 <= 9) ?
00205             ('0' + digit2) : ('A' + digit2 - 0xA);
00206     }
00207
00208     /* Terminate string for output */
00209     newstring[i++] = '\n';
00210     newstring[i++] = '\0';
00211
00212     memcpy (glyphstring, newstring, i);
00213
00214     return;
00215 }
00216
00217
00218
00219 /**
00220  @brief Superimpose a double-width dashed combining circle on a glyph bitmap.
00221
00222  @param[in,out] glyphstring A double-width glyph, 16x16 pixels.
00223 */

```

```

00224 void
00225 add_double_circle (char *glyphstring, int offset)
00226 {
00227     char newstring[256];
00228     /* Circle hex string pattern is "00000008000024004200240000000000" */
00229     /* For double diacritical glyphs (offset = -8) */
00230     /* Combining circle is left-justified. */
00231     char circle08[64]={0x0,0x0,0x0,0x0, /* row 1 */
00232                        0x0,0x0,0x0,0x0, /* row 2 */
00233                        0x0,0x0,0x0,0x0, /* row 3 */
00234                        0x0,0x0,0x0,0x0, /* row 4 */
00235                        0x0,0x0,0x0,0x0, /* row 5 */
00236                        0x0,0x0,0x0,0x0, /* row 6 */
00237                        0x2,0x4,0x0,0x0, /* row 7 */
00238                        0x0,0x0,0x0,0x0, /* row 8 */
00239                        0x4,0x2,0x0,0x0, /* row 9 */
00240                        0x0,0x0,0x0,0x0, /* row 10 */
00241                        0x2,0x4,0x0,0x0, /* row 11 */
00242                        0x0,0x0,0x0,0x0, /* row 12 */
00243                        0x0,0x0,0x0,0x0, /* row 13 */
00244                        0x0,0x0,0x0,0x0, /* row 14 */
00245                        0x0,0x0,0x0,0x0, /* row 15 */
00246                        0x0,0x0,0x0,0x0}; /* row 16 */
00247
00248     /* For all other combining glyphs (offset = -16) */
00249     /* Combining circle is centered in 16 columns. */
00250     char circle16[64]={0x0,0x0,0x0,0x0, /* row 1 */
00251                        0x0,0x0,0x0,0x0, /* row 2 */
00252                        0x0,0x0,0x0,0x0, /* row 3 */
00253                        0x0,0x0,0x0,0x0, /* row 4 */
00254                        0x0,0x0,0x0,0x0, /* row 5 */
00255                        0x0,0x0,0x0,0x0, /* row 6 */
00256                        0x0,0x2,0x4,0x0, /* row 7 */
00257                        0x0,0x0,0x0,0x0, /* row 8 */
00258                        0x0,0x4,0x2,0x0, /* row 9 */
00259                        0x0,0x0,0x0,0x0, /* row 10 */
00260                        0x0,0x2,0x4,0x0, /* row 11 */
00261                        0x0,0x0,0x0,0x0, /* row 12 */
00262                        0x0,0x0,0x0,0x0, /* row 13 */
00263                        0x0,0x0,0x0,0x0, /* row 14 */
00264                        0x0,0x0,0x0,0x0, /* row 15 */
00265                        0x0,0x0,0x0,0x0}; /* row 16 */
00266
00267     char *circle; /* points into circle16 or circle08 */
00268
00269     int digit1, digit2; /* corresponding digits in each string */
00270
00271     int i; /* index variables */
00272
00273     /*
00274      * Determine if combining circle is left-justified (offset = -8)
00275      * or centered (offset = -16).
00276      */
00277     circle = (offset >= -8) ? circle08 : circle16;
00278
00279     /* for each character position, OR the corresponding circle glyph value */
00280     for (i = 0; i < 64; i++) {
00281         glyphstring[i] = toupper (glyphstring[i]);
00282
00283         /* Convert ASCII character to a hexadecimal integer */
00284         digit1 = (glyphstring[i] <= '9') ?
00285                 (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);
00286
00287         /* Superimpose dashed circle */
00288         digit2 = digit1 | circle[i];
00289
00290         /* Convert hexadecimal integer to an ASCII character */
00291         newstring[i] = (digit2 <= 9) ?
00292                 ('0' + digit2) : ('A' + digit2 - 0xA);
00293     }
00294
00295     /* Terminate string for output */
00296     newstring[i++] = '\n';
00297     newstring[i++] = '\0';
00298
00299     memcpy (glyphstring, newstring, i);
00300
00301     return;

```

```
00305 }  
00306
```

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](#) 0xF0E70
Start of Pikto code point range.
- #define [PIKTO_END](#) 0xF11EF
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

Copyright (C) 2013, 2017 Paul Hardy.

All glyphs are treated as 16 pixels high, and can be 8, 16, 24, or 32 pixels wide (resulting in widths of 1, 2, 3, or 4, respectively).

Definition in file [unigenwidth.c](#).

5.31.2 Macro Definition Documentation

5.31.2.1 MAXSTRING

```
#define MAXSTRING 256
```

Maximum input line length - 1.

Definition at line [50](#) of file [unigenwidth.c](#).

5.31.2.2 PIKTO__END

```
#define PIKTO__END 0xF11EF
```

End of Pikto code point range.

Definition at line [54](#) 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 [56](#) of file [unigenwidth.c](#).

5.31.2.4 PIKTO__START

```
#define PIKTO__START 0xF0E70
```

Start of Pikto code point range.

Definition at line [53](#) 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 67 of file `unigenwidth.c`.

```

00068 {
00069
00070     int i; /* loop variable */
00071
00072     char teststring[MAXSTRING];
00073     unsigned loc;
00074     char *gstart;
00075
00076     char glyph_width[0x20000];
00077     char pikto_width[PIKTO_SIZE];
00078
00079     FILE *infilep;
00080
00081     if (argc != 3) {
00082         fprintf(stderr, "\n\nUsage: %s <unifont.hex> <combining.txt>\n\n", argv[0]);
00083         exit (EXIT_FAILURE);
00084     }
00085
00086     /*
00087      * Read the collection of hex glyphs.
00088      */
00089     if ((infilep = fopen (argv[1], "r")) == NULL) {
00090         fprintf(stderr, "ERROR - hex input file %s not found.\n\n", argv[1]);
00091         exit (EXIT_FAILURE);
00092     }
00093
00094     /* Flag glyph as non-existent until found. */
00095     memset (glyph_width, -1, 0x20000 * sizeof (char));
00096     memset (pikto_width, -1, (PIKTO_SIZE) * sizeof (char));
00097
00098     teststring[MAXSTRING-1] = '\0';
00099     while (fgets (teststring, MAXSTRING-1, infilep) != NULL) {
00100         sscanf (teststring, "%X:%*s", &loc);
00101         if (loc < 0x20000) {
00102             gstart = strchr (teststring, ':') + 1;
00103             /*
00104              * 16 rows per glyph, 2 ASCII hexadecimal digits per byte,
00105              * so divide number of digits by 32 (shift right 5 bits).
00106              */
00107             glyph_width[loc] = (strlen (gstart) - 1) » 5;
00108         }
00109         else if ((loc >= PIKTO_START) && (loc <= PIKTO_END)) {
00110             gstart = strchr (teststring, ':') + 1;
00111             pikto_width[loc - PIKTO_START] = strlen (gstart) <= 34 ? 1 : 2;
00112         }
00113     }
00114
00115     fclose (infilep);
00116
00117     /*
00118      * Now read the combining character code points. These have width of 0.
00119      */
00120     if ((infilep = fopen (argv[2], "r")) == NULL) {
00121         fprintf(stderr, "ERROR - combining characters file %s not found.\n\n", argv[2]);
00122         exit (EXIT_FAILURE);
00123     }
00124
00125     while (fgets (teststring, MAXSTRING-1, infilep) != NULL) {
00126         sscanf (teststring, "%X:%*s", &loc);
00127         if (loc < 0x20000) glyph_width[loc] = 0;
00128     }
00129
00130     fclose (infilep);

```

```

00131
00132 /*
00133    Code Points with Unusual Properties (Unicode Standard, Chapter 4).
00134
00135    As of Unifont 10.0.04, use the widths in the "nonprinting.hex"
00136    files. If an application is smart enough to know how to handle
00137    these special cases, it will not render the "nonprinting" glyph
00138    and will treat the code point as being zero-width.
00139 */
00140 // glyph_width[0]=0; /* NULL character */
00141 // for (i = 0x0001; i <= 0x001F; i++) glyph_width[i]=-1; /* Control Characters */
00142 // for (i = 0x007F; i <= 0x009F; i++) glyph_width[i]=-1; /* Control Characters */
00143
00144 // glyph_width[0x034F]=0; /* combining grapheme joiner */
00145 // glyph_width[0x180B]=0; /* Mongolian free variation selector one */
00146 // glyph_width[0x180C]=0; /* Mongolian free variation selector two */
00147 // glyph_width[0x180D]=0; /* Mongolian free variation selector three */
00148 // glyph_width[0x180E]=0; /* Mongolian vowel separator */
00149 // glyph_width[0x200B]=0; /* zero width space */
00150 // glyph_width[0x200C]=0; /* zero width non-joiner */
00151 // glyph_width[0x200D]=0; /* zero width joiner */
00152 // glyph_width[0x200E]=0; /* left-to-right mark */
00153 // glyph_width[0x200F]=0; /* right-to-left mark */
00154 // glyph_width[0x202A]=0; /* left-to-right embedding */
00155 // glyph_width[0x202B]=0; /* right-to-left embedding */
00156 // glyph_width[0x202C]=0; /* pop directional formatting */
00157 // glyph_width[0x202D]=0; /* left-to-right override */
00158 // glyph_width[0x202E]=0; /* right-to-left override */
00159 // glyph_width[0x2060]=0; /* word joiner */
00160 // glyph_width[0x2061]=0; /* function application */
00161 // glyph_width[0x2062]=0; /* invisible times */
00162 // glyph_width[0x2063]=0; /* invisible separator */
00163 // glyph_width[0x2064]=0; /* invisible plus */
00164 // glyph_width[0x206A]=0; /* inhibit symmetric swapping */
00165 // glyph_width[0x206B]=0; /* activate symmetric swapping */
00166 // glyph_width[0x206C]=0; /* inhibit arabic form shaping */
00167 // glyph_width[0x206D]=0; /* activate arabic form shaping */
00168 // glyph_width[0x206E]=0; /* national digit shapes */
00169 // glyph_width[0x206F]=0; /* nominal digit shapes */
00170
00171 // /* Variation Selector-1 to Variation Selector-16 */
00172 // for (i = 0xFE00; i <= 0xFE0F; i++) glyph_width[i] = 0;
00173
00174 // glyph_width[0xFEFF]=0; /* zero width no-break space */
00175 // glyph_width[0xFF9]=0; /* interlinear annotation anchor */
00176 // glyph_width[0xFFA]=0; /* interlinear annotation separator */
00177 // glyph_width[0xFFB]=0; /* interlinear annotation terminator */
00178 /*
00179    Let glyph widths represent 0xFFFC (object replacement character)
00180    and 0xFFFD (replacement character).
00181 */
00182
00183 /*
00184    Hangul Jamo:
00185
00186    Leading Consonant (Choseong): leave spacing as is.
00187
00188    Hangul Choseong Filler (U+115F): set width to 2.
00189
00190    Hangul Jungseong Filler, Hangul Vowel (Jungseong), and
00191    Final Consonant (Jongseong): set width to 0, because these
00192    combine with the leading consonant as one composite syllabic
00193    glyph. As of Unicode 5.2, the Hangul Jamo block (U+1100..U+11FF)
00194    is completely filled.
00195 */
00196 // for (i = 0x1160; i <= 0x11FF; i++) glyph_width[i]=0; /* Vowels & Final Consonants */
00197
00198 /*
00199    Private Use Area -- the width is undefined, but likely
00200    to be 2 charcells wide either from a graphic glyph or
00201    from a four-digit hexadecimal glyph representing the
00202    code point. Therefore if any PUA glyph does not have
00203    a non-zero width yet, assign it a default width of 2.
00204    The Unicode Standard allows giving PUA characters
00205    default property values; see for example The Unicode
00206    Standard Version 5.0, p. 91. This same default is
00207    used for higher plane PUA code points below.
00208 */
00209 // for (i = 0xE000; i <= 0xF8FF; i++) {
00210 //     if (glyph_width[i] == 0) glyph_width[i]=2;
00211 // }

```



```

00212
00213 /*
00214  <not a character>
00215 */
00216 for (i = 0xFDD0; i <= 0xFDEF; i++) glyph_width[i] = -1;
00217 glyph_width[0xFFFF] = -1; /* Byte Order Mark */
00218 glyph_width[0xFFFF] = -1; /* Byte Order Mark */
00219
00220 /* Surrogate Code Points */
00221 for (i = 0xD800; i <= 0xDFFF; i++) glyph_width[i] = -1;
00222
00223 /* CJK Code Points */
00224 for (i = 0x4E00; i <= 0x9FFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00225 for (i = 0x3400; i <= 0x4DBF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00226 for (i = 0xF900; i <= 0xFAFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00227
00228 /*
00229  Now generate the output file.
00230 */
00231 printf ("/*\n");
00232 printf (" wcswidth and wcswidth functions, as per IEEE 1003.1-2008\n");
00233 printf (" System Interfaces, pp. 2241 and 2251.\n\n");
00234 printf (" Author: Paul Hardy, 2013\n\n");
00235 printf (" Copyright (c) 2013 Paul Hardy\n\n");
00236 printf (" LICENSE:\n");
00237 printf ("\n");
00238 printf (" This program is free software: you can redistribute it and/or modify\n");
00239 printf (" it under the terms of the GNU General Public License as published by\n");
00240 printf (" the Free Software Foundation, either version 2 of the License, or\n");
00241 printf (" (at your option) any later version.\n");
00242 printf ("\n");
00243 printf (" This program is distributed in the hope that it will be useful,\n");
00244 printf (" but WITHOUT ANY WARRANTY; without even the implied warranty of\n");
00245 printf (" MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the\n");
00246 printf (" GNU General Public License for more details.\n");
00247 printf ("\n");
00248 printf (" You should have received a copy of the GNU General Public License\n");
00249 printf (" along with this program. If not, see <http://www.gnu.org/licenses/>.\n");
00250 printf ("*/\n\n");
00251
00252 printf ("#include <wchar.h>\n\n");
00253 printf ("/* Definitions for Pkito CSUR Private Use Area glyphs */\n");
00254 printf ("#define PIKTO_START\t0x%06X\n", PIKTO_START);
00255 printf ("#define PIKTO_END\t0x%06X\n", PIKTO_END);
00256 printf ("#define PIKTO_SIZE\t(PIKTO_END - PIKTO_START + 1)\n");
00257 printf ("\n\n");
00258 printf ("/* wcswidth -- return charcell positions of one code point */\n");
00259 printf ("inline int nwcswidth (wchar_t wc)\n{\n");
00260 printf (" return (wcswidth (&wc, 1));\n");
00261 printf ("}\n");
00262 printf ("\n\n");
00263 printf ("int nwcswidth (const wchar_t *pwcs, size_t n)\n{\n\n");
00264 printf (" int i; /* loop variable */\n");
00265 printf (" unsigned codept; /* Unicode code point of current character */\n");
00266 printf (" unsigned plane; /* Unicode plane, 0x00..0x10 */\n");
00267 printf (" unsigned lower17; /* lower 17 bits of Unicode code point */\n");
00268 printf (" unsigned lower16; /* lower 16 bits of Unicode code point */\n");
00269 printf (" int lowpt, midpt, highpt; /* for binary searching in plane1zeroes[] */\n");
00270 printf (" int found; /* for binary searching in plane1zeroes[] */\n");
00271 printf (" int totalwidth; /* total width of string, in charcells (1 or 2/glyph) */\n");
00272 printf (" int illegalchar; /* Whether or not this code point is illegal */\n");
00273 putchar ('\n');
00274
00275 /*
00276  Print the glyph_width[] array for glyphs widths in the
00277  Basic Multilingual Plane (Plane 0).
00278 */
00279 printf (" char glyph_width[0x20000] = {\n");
00280 for (i = 0; i < 0x10000; i++) {
00281     if ((i & 0x1F) == 0)
00282         printf ("\n /* U+%04X */ ", i);
00283     printf ("%d", glyph_width[i]);
00284 }
00285 for (i = 0x10000; i < 0x20000; i++) {
00286     if ((i & 0x1F) == 0)
00287         printf ("\n /* U+%06X */ ", i);
00288     printf ("%d", glyph_width[i]);
00289     if (i < 0x1FFFF) putchar (',' );
00290 }
00291 printf ("\n }; \n\n");
00292

```

```

00293  /*
00294  Print the pikto_width[] array for Pikto glyph widths.
00295  */
00296  printf (" char pikto_width[PIKTO_SIZE] = {");
00297  for (i = 0; i < PIKTO_SIZE; i++) {
00298      if ((i & 0x1F) == 0)
00299          printf ("\n /* U+%06X */ ", PIKTO_START + i);
00300      printf ("%d", pikto_width[i]);
00301      if ((PIKTO_START + i) < PIKTO_END) putchar (',' );
00302  }
00303  printf ("\n }; \n\n");
00304
00305  /*
00306  Execution part of wcswidth.
00307  */
00308  printf ("\n");
00309  printf (" illegalchar = totalwidth = 0; \n");
00310  printf (" for (i = 0; !illegalchar && i < n; i++) { \n");
00311  printf ("     codept = pwcs[i]; \n");
00312  printf ("     plane = codept >> 16; \n");
00313  printf ("     lower17 = codept & 0x1FFFF; \n");
00314  printf ("     lower16 = codept & 0xFFFF; \n");
00315  printf ("     if (plane < 2) { /* the most common case */ \n");
00316  printf ("         if (glyph_width[lower17] < 0) illegalchar = 1; \n");
00317  printf ("         else totalwidth += glyph_width[lower17]; \n");
00318  printf ("     } \n");
00319  printf ("     else { /* a higher plane or beyond Unicode range */ \n");
00320  printf ("         if ((lower16 == 0xFFFE) || (lower16 == 0xFFFF)) { \n");
00321  printf ("             illegalchar = 1; \n");
00322  printf ("         } \n");
00323  printf ("         else if (plane < 4) { /* Ideographic Plane */ \n");
00324  printf ("             totalwidth += 2; /* Default ideographic width */ \n");
00325  printf ("         } \n");
00326  printf ("         else if (plane == 0x0F) { /* CSUR Private Use Area */ \n");
00327  printf ("             if (lower16 <= 0x0E6F) { /* Kinya */ \n");
00328  printf ("                 totalwidth++; /* all Kinya syllables have width 1 */ \n");
00329  printf ("             } \n");
00330  printf ("             else if (lower16 <= (PIKTO_END & 0xFFFF)) { /* Pikto */ \n");
00331  printf ("                 if (pikto_width[lower16 - (PIKTO_START & 0xFFFF)] < 0) illegalchar = 1; \n");
00332  printf ("                 else totalwidth += pikto_width[lower16 - (PIKTO_START & 0xFFFF)]; \n");
00333  printf ("             } \n");
00334  printf ("         } \n");
00335  printf ("         else if (plane > 0x10) { \n");
00336  printf ("             illegalchar = 1; \n");
00337  printf ("         } \n");
00338  printf ("         /* Other non-printing in higher planes; return -1 as per IEEE 1003.1-2008. */ \n");
00339  printf ("     } \n");
00340  printf ("     else if (/* language tags */ \n");
00341  printf ("         codept == 0x0E0001 || (codept >= 0x0E0020 && codept <= 0x0E007F) || \n");
00342  printf ("         /* variation selectors, 0x0E0100..0x0E01EF */ \n");
00343  printf ("         (codept >= 0x0E0100 && codept <= 0x0E01EF)) { \n");
00344  printf ("             illegalchar = 1; \n");
00345  printf ("         } \n");
00346  printf ("     } \n");
00347  printf ("     /* Unicode plane 0x02..0x10 printing character */ \n");
00348  printf ("     */ \n");
00349  printf ("     else { \n");
00350  printf ("         illegalchar = 1; /* code is not in font */ \n");
00351  printf ("     } \n");
00352  printf (" } \n");
00353  printf (" } \n");
00354  printf (" if (illegalchar) totalwidth = -1; \n");
00355  printf (" \n");
00356  printf (" return (totalwidth); \n");
00357  printf (" \n");
00358  printf (" } \n");
00359
00360  exit (EXIT_SUCCESS);
00361 }

```

5.32 unigenwidth.c

[Go to the documentation of this file.](#)

```

00001 /**
00002  @file unigenwidth.c

```

```

00003
00004  @brief unigenwidth - IEEE 1003.1-2008 setup to calculate
00005          wchar_t string widths
00006
00007  @author Paul Hardy.
00008
00009  @copyright Copyright (C) 2013, 2017 Paul Hardy.
00010
00011  All glyphs are treated as 16 pixels high, and can be
00012  8, 16, 24, or 32 pixels wide (resulting in widths of
00013  1, 2, 3, or 4, respectively).
00014 */
00015 /*
00016  LICENSE:
00017
00018   This program is free software: you can redistribute it and/or modify
00019   it under the terms of the GNU General Public License as published by
00020   the Free Software Foundation, either version 2 of the License, or
00021   (at your option) any later version.
00022
00023   This program is distributed in the hope that it will be useful,
00024   but WITHOUT ANY WARRANTY; without even the implied warranty of
00025   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00026   GNU General Public License for more details.
00027
00028   You should have received a copy of the GNU General Public License
00029   along with this program. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031
00032 /*
00033   20 June 2017 [Paul Hardy]:
00034   - Now handles glyphs that are 24 or 32 pixels wide.
00035
00036   8 July 2017 [Paul Hardy]:
00037   - Modifies sscanf format strings to ignore second field after
00038   the ":" field separator, newly added to "combining.txt" files
00039   and already present in ".hex" files.
00040
00041   6 September 2025 [Paul Hardy]:
00042   - Changed loc from "int" to "unsigned" for compatibility with
00043   sscanf definition.
00044 */
00045
00046 #include <stdio.h>
00047 #include <stdlib.h>
00048 #include <string.h>
00049
00050 #define MAXSTRING 256 ///< Maximum input line length - 1.
00051
00052 /* Definitions for Pikto in Plane 15 */
00053 #define PIKTO_START 0x0F0E70 ///< Start of Pikto code point range.
00054 #define PIKTO_END 0x0F11EF ///< End of Pikto code point range.
00055 /** Number of code points in Pikto range. */
00056 #define PIKTO_SIZE (PIKTO_END - PIKTO_START + 1)
00057
00058
00059 /**
00060  @brief The main function.
00061
00062  @param[in] argc The count of command line arguments.
00063  @param[in] argv Pointer to array of command line arguments.
00064  @return This program exits with status EXIT_SUCCESS.
00065 */
00066 int
00067 main (int argc, char **argv)
00068 {
00069
00070   int i; /* loop variable */
00071
00072   char teststring[MAXSTRING];
00073   unsigned loc;
00074   char *gstart;
00075
00076   char glyph_width[0x20000];
00077   char pikto_width[PIKTO_SIZE];
00078
00079   FILE *infilep;
00080
00081   if (argc != 3) {
00082     fprintf (stderr, "\n\nUsage: %s <unifont.hex> <combining.txt>\n\n", argv[0]);
00083     exit (EXIT_FAILURE);

```

```

00084 }
00085
00086 /*
00087  Read the collection of hex glyphs.
00088 */
00089 if ((infilep = fopen (argv[1], "r")) == NULL) {
00090     fprintf (stderr, "ERROR - hex input file %s not found.\n\n", argv[1]);
00091     exit (EXIT_FAILURE);
00092 }
00093
00094 /* Flag glyph as non-existent until found. */
00095 memset (glyph_width, -1, 0x20000 * sizeof (char));
00096 memset (pikto_width, -1, (PIKTO_SIZE) * sizeof (char));
00097
00098 teststring[MAXSTRING-1] = '\0';
00099 while (fgets (teststring, MAXSTRING-1, infilep) != NULL) {
00100     sscanf (teststring, "%X:%*s", &loc);
00101     if (loc < 0x20000) {
00102         gstart = strchr (teststring, ':') + 1;
00103         /*
00104          16 rows per glyph, 2 ASCII hexadecimal digits per byte,
00105          so divide number of digits by 32 (shift right 5 bits).
00106          */
00107         glyph_width[loc] = (strlen (gstart) - 1) » 5;
00108     }
00109     else if ((loc >= PIKTO_START) && (loc <= PIKTO_END)) {
00110         gstart = strchr (teststring, ':') + 1;
00111         pikto_width[loc - PIKTO_START] = strlen (gstart) <= 34 ? 1 : 2;
00112     }
00113 }
00114
00115 fclose (infilep);
00116
00117 /*
00118  Now read the combining character code points. These have width of 0.
00119 */
00120 if ((infilep = fopen (argv[2], "r")) == NULL) {
00121     fprintf (stderr, "ERROR - combining characters file %s not found.\n\n", argv[2]);
00122     exit (EXIT_FAILURE);
00123 }
00124
00125 while (fgets (teststring, MAXSTRING-1, infilep) != NULL) {
00126     sscanf (teststring, "%X:%*s", &loc);
00127     if (loc < 0x20000) glyph_width[loc] = 0;
00128 }
00129
00130 fclose (infilep);
00131
00132 /*
00133  Code Points with Unusual Properties (Unicode Standard, Chapter 4).
00134
00135  As of Unifont 10.0.04, use the widths in the *-nonprinting.hex"
00136  files. If an application is smart enough to know how to handle
00137  these special cases, it will not render the "nonprinting" glyph
00138  and will treat the code point as being zero-width.
00139 */
00140 // glyph_width[0]=0; /* NULL character */
00141 // for (i = 0x0001; i <= 0x001F; i++) glyph_width[i]=-1; /* Control Characters */
00142 // for (i = 0x007F; i <= 0x009F; i++) glyph_width[i]=-1; /* Control Characters */
00143
00144 // glyph_width[0x034F]=0; /* combining grapheme joiner */
00145 // glyph_width[0x180B]=0; /* Mongolian free variation selector one */
00146 // glyph_width[0x180C]=0; /* Mongolian free variation selector two */
00147 // glyph_width[0x180D]=0; /* Mongolian free variation selector three */
00148 // glyph_width[0x180E]=0; /* Mongolian vowel separator */
00149 // glyph_width[0x200B]=0; /* zero width space */
00150 // glyph_width[0x200C]=0; /* zero width non-joiner */
00151 // glyph_width[0x200D]=0; /* zero width joiner */
00152 // glyph_width[0x200E]=0; /* left-to-right mark */
00153 // glyph_width[0x200F]=0; /* right-to-left mark */
00154 // glyph_width[0x202A]=0; /* left-to-right embedding */
00155 // glyph_width[0x202B]=0; /* right-to-left embedding */
00156 // glyph_width[0x202C]=0; /* pop directional formatting */
00157 // glyph_width[0x202D]=0; /* left-to-right override */
00158 // glyph_width[0x202E]=0; /* right-to-left override */
00159 // glyph_width[0x2060]=0; /* word joiner */
00160 // glyph_width[0x2061]=0; /* function application */
00161 // glyph_width[0x2062]=0; /* invisible times */
00162 // glyph_width[0x2063]=0; /* invisible separator */
00163 // glyph_width[0x2064]=0; /* invisible plus */
00164 // glyph_width[0x206A]=0; /* inhibit symmetric swapping */

```

```

00165 // glyph_width[0x206B]=0; /* activate symmetric swapping */
00166 // glyph_width[0x206C]=0; /* inhibit arabic form shaping */
00167 // glyph_width[0x206D]=0; /* activate arabic form shaping */
00168 // glyph_width[0x206E]=0; /* national digit shapes */
00169 // glyph_width[0x206F]=0; /* nominal digit shapes */
00170
00171 // /* Variation Selector-1 to Variation Selector-16 */
00172 // for (i = 0xFE00; i <= 0xFE0F; i++) glyph_width[i] = 0;
00173
00174 // glyph_width[0xFEFF]=0; /* zero width no-break space */
00175 // glyph_width[0xFFFF9]=0; /* interlinear annotation anchor */
00176 // glyph_width[0xFFFFA]=0; /* interlinear annotation separator */
00177 // glyph_width[0xFFFFB]=0; /* interlinear annotation terminator */
00178 /*
00179     Let glyph widths represent 0xFFFC (object replacement character)
00180     and 0xFFFD (replacement character).
00181 */
00182
00183 /*
00184     Hangul Jamo:
00185
00186     Leading Consonant (Choseong): leave spacing as is.
00187
00188     Hangul Choseong Filler (U+115F): set width to 2.
00189
00190     Hangul Jungseong Filler, Hangul Vowel (Jungseong), and
00191     Final Consonant (Jongseong): set width to 0, because these
00192     combine with the leading consonant as one composite syllabic
00193     glyph. As of Unicode 5.2, the Hangul Jamo block (U+1100..U+11FF)
00194     is completely filled.
00195 */
00196 // for (i = 0x1160; i <= 0x11FF; i++) glyph_width[i]=0; /* Vowels & Final Consonants */
00197
00198 /*
00199     Private Use Area -- the width is undefined, but likely
00200     to be 2 charcells wide either from a graphic glyph or
00201     from a four-digit hexadecimal glyph representing the
00202     code point. Therefore if any PUA glyph does not have
00203     a non-zero width yet, assign it a default width of 2.
00204     The Unicode Standard allows giving PUA characters
00205     default property values; see for example The Unicode
00206     Standard Version 5.0, p. 91. This same default is
00207     used for higher plane PUA code points below.
00208 */
00209 // for (i = 0xE000; i <= 0xF8FF; i++) {
00210 //     if (glyph_width[i] == 0) glyph_width[i]=2;
00211 // }
00212
00213 /*
00214     <not a character>
00215 */
00216 for (i = 0xFDD0; i <= 0xFDEF; i++) glyph_width[i] = -1;
00217 glyph_width[0xFFFFE] = -1; /* Byte Order Mark */
00218 glyph_width[0xFFFFF] = -1; /* Byte Order Mark */
00219
00220 /* Surrogate Code Points */
00221 for (i = 0xD800; i <= 0xDFFF; i++) glyph_width[i]=-1;
00222
00223 /* CJK Code Points */
00224 for (i = 0x4E00; i <= 0x9FFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00225 for (i = 0x3400; i <= 0x4DBF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00226 for (i = 0xF900; i <= 0xFAFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00227
00228 /*
00229     Now generate the output file.
00230 */
00231 printf ("/*\n");
00232 printf (" wewidth and wcswidth functions, as per IEEE 1003.1-2008\n");
00233 printf (" System Interfaces, pp. 2241 and 2251.\n\n");
00234 printf (" Author: Paul Hardy, 2013\n\n");
00235 printf (" Copyright (c) 2013 Paul Hardy\n\n");
00236 printf (" LICENSE:\n");
00237 printf ("\n");
00238 printf (" This program is free software: you can redistribute it and/or modify\n");
00239 printf (" it under the terms of the GNU General Public License as published by\n");
00240 printf (" the Free Software Foundation, either version 2 of the License, or\n");
00241 printf (" (at your option) any later version.\n");
00242 printf ("\n");
00243 printf (" This program is distributed in the hope that it will be useful,\n");
00244 printf (" but WITHOUT ANY WARRANTY; without even the implied warranty of\n");
00245 printf (" MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the\n");

```

```

00246 printf ("    GNU General Public License for more details.\n");
00247 printf ("\n");
00248 printf ("    You should have received a copy of the GNU General Public License\n");
00249 printf ("    along with this program. If not, see <http://www.gnu.org/licenses/>.\n");
00250 printf ("*/\n\n");
00251
00252 printf ("#include <wchar.h>\n\n");
00253 printf ("/* Definitions for Pikto CSUR Private Use Area glyphs */\n");
00254 printf ("#define PIKTO_START\t0x%06X\n", PIKTO_START);
00255 printf ("#define PIKTO_END\t0x%06X\n", PIKTO_END);
00256 printf ("#define PIKTO_SIZE\t(PIKTO_END - PIKTO_START + 1)\n");
00257 printf ("\n\n");
00258 printf ("/* wwidth -- return charcell positions of one code point */\n");
00259 printf ("inline int nwidth (wchar_t wc)\n{\n");
00260 printf ("    return (wcswidth (&wc, 1));\n");
00261 printf ("}\n");
00262 printf ("\n\n");
00263 printf ("int nwidth (const wchar_t *pwcs, size_t n)\n{\n\n");
00264 printf ("    int i;                /* loop variable */\n");
00265 printf ("    unsigned codept;      /* Unicode code point of current character */\n");
00266 printf ("    unsigned plane;       /* Unicode plane, 0x00..0x10 */\n");
00267 printf ("    unsigned lower17;     /* lower 17 bits of Unicode code point */\n");
00268 printf ("    unsigned lower16;     /* lower 16 bits of Unicode code point */\n");
00269 printf ("    int lowpt, midpt, highpt; /* for binary searching in plane zeroes[] */\n");
00270 printf ("    int found;            /* for binary searching in plane zeroes[] */\n");
00271 printf ("    int totalwidth;       /* total width of string, in charcells (1 or 2/glyph) */\n");
00272 printf ("    int illegalchar;      /* Whether or not this code point is illegal */\n");
00273 putchar ('\n');
00274
00275 /*
00276  Print the glyph_width[] array for glyphs widths in the
00277  Basic Multilingual Plane (Plane 0).
00278 */
00279 printf ("    char glyph_width[0x20000] = {\n");
00280 for (i = 0; i < 0x10000; i++) {
00281     if ((i & 0x1F) == 0)
00282         printf ("        /* U+%04X */ ", i);
00283     printf ("%d", glyph_width[i]);
00284 }
00285 for (i = 0x10000; i < 0x20000; i++) {
00286     if ((i & 0x1F) == 0)
00287         printf ("        /* U+%06X */ ", i);
00288     printf ("%d", glyph_width[i]);
00289     if (i < 0x1FFFF) putchar (',' );
00290 }
00291 printf ("    };\n\n");
00292
00293 /*
00294  Print the pikto_width[] array for Pikto glyph widths.
00295 */
00296 printf ("    char pikto_width[PIKTO_SIZE] = {\n");
00297 for (i = 0; i < PIKTO_SIZE; i++) {
00298     if ((i & 0x1F) == 0)
00299         printf ("        /* U+%06X */ ", PIKTO_START + i);
00300     printf ("%d", pikto_width[i]);
00301     if ((PIKTO_START + i) < PIKTO_END) putchar (',' );
00302 }
00303 printf ("    };\n\n");
00304
00305 /*
00306  Execution part of wcswidth.
00307 */
00308 printf ("\n");
00309 printf ("    illegalchar = totalwidth = 0;\n");
00310 printf ("    for (i = 0; !illegalchar && i < n; i++) {\n");
00311 printf ("        codept = pwcs[i];\n");
00312 printf ("        plane = codept >> 16;\n");
00313 printf ("        lower17 = codept & 0x1FFFF;\n");
00314 printf ("        lower16 = codept & 0xFFFF;\n");
00315 printf ("        if (plane < 2) { /* the most common case */\n");
00316 printf ("            if (glyph_width[lower17] < 0) illegalchar = 1;\n");
00317 printf ("            else totalwidth += glyph_width[lower17];\n");
00318 printf ("        };\n");
00319 printf ("        else { /* a higher plane or beyond Unicode range */\n");
00320 printf ("            if ((lower16 == 0xFFFE) || (lower16 == 0xFFFF)) {\n");
00321 printf ("                illegalchar = 1;\n");
00322 printf ("            };\n");
00323 printf ("            else if (plane < 4) { /* Ideographic Plane */\n");
00324 printf ("                totalwidth += 2; /* Default ideographic width */\n");
00325 printf ("            };\n");
00326 printf ("            else if (plane == 0x0F) { /* CSUR Private Use Area */\n");

```

```

00327 printf ("        if (lower16 <= 0x0E6F) { /* Kinya */\n");
00328 printf ("            totalwidth++; /* all Kinya syllables have width 1 */\n");
00329 printf ("        }\n");
00330 printf ("        else if (lower16 <= (PIKTO_END & 0xFFFF)) { /* Pikto */\n");
00331 printf ("            if (pikto_width[lower16 - (PIKTO_START & 0xFFFF)] < 0) illegalchar = 1;\n");
00332 printf ("            else totalwidth += pikto_width[lower16 - (PIKTO_START & 0xFFFF)];\n");
00333 printf ("        }\n");
00334 printf ("    }\n");
00335 printf ("    else if (plane > 0x10) {\n");
00336 printf ("        illegalchar = 1;\n");
00337 printf ("    }\n");
00338 printf ("    /* Other non-printing in higher planes; return -1 as per IEEE 1003.1-2008. */\n");
00339 printf ("    else if (/* language tags */\n");
00340 printf ("        codept == 0x0E0001 || (codept >= 0x0E0020 && codept <= 0x0E007F) ||\n");
00341 printf ("        /* variation selectors, 0x0E0100..0x0E01EF */\n");
00342 printf ("        (codept >= 0x0E0100 && codept <= 0x0E01EF)) {\n");
00343 printf ("        illegalchar = 1;\n");
00344 printf ("    }\n");
00345 printf ("    /*\n");
00346 printf ("        Unicode plane 0x02..0x10 printing character\n");
00347 printf ("        */\n");
00348 printf ("    else {\n");
00349 printf ("        illegalchar = 1; /* code is not in font */\n");
00350 printf ("    }\n");
00351 printf ("    }\n");
00352 printf ("    }\n");
00353 printf ("    }\n");
00354 printf ("    if (illegalchar) totalwidth = -1;\n");
00355 printf ("    }\n");
00356 printf ("    return (totalwidth);\n");
00357 printf ("    }\n");
00358 printf ("}\n");
00359
00360 exit (EXIT_SUCCESS);
00361 }

```

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 initial 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

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-ARAE, ARAEA, ARAEA-EO,
00394 /* 0x3B8: */ 1, 2, 1, // U+11A0..U+11A2-->[63..65] ARAEA-U, ARAEA-ISSANGARAE,
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) {
00414     cho_variation = -1;
00415 }
00416 else {
00417     cho_variation = choseong_var[jungseong];
00418     if (choseong >= 0 && jongseong >= 0 && cho_variation < 3)
00419         cho_variation += 3;
00420 }
00421
00422
00423 return cho_variation;
00424 }

```

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](#).

```
00638     {
00639     int i;
00640
00641     for (i = 0; i < 16; i++)
00642         combined_glyph [i] = glyph1 [i] | glyph2 [i];
00643
00644     return;
00645 }
```

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](#).

```
00789     {
00790
00791     int i; /* Loop variable. */
00792     int cho_num, jung_num, jong_num;
```

```

00793 int cho_group, jung_group, jong_group;
00794 int cho_index, jung_index, jong_index;
00795
00796 unsigned tmp_glyph[16]; /* Hold shifted jongsung for wide vertical vowel. */
00797
00798 int cho_variation (int choseong, int jungseong, int jongseong);
00799
00800 void combine_glyphs (unsigned *glyph1, unsigned *glyph2,
00801                    unsigned *combined_glyph);
00802
00803 /* Choose a blank glyph for each syllable by default. */
00804 cho_index = jung_index = jong_index = 0x000;
00805
00806 /*
00807  * Convert Unicode code points to jamo sequence number
00808  * of each letter, or -1 if letter is not in valid range.
00809  */
00810 if (cho >= 0x1100 && cho <= 0x115E)
00811     cho_num = cho - CHO_UNICODE_START;
00812 else if (cho >= CHO_EXT_A_UNICODE_START &&
00813         cho < (CHO_EXT_A_UNICODE_START + NCHO_EXT_A))
00814     cho_num = cho - CHO_EXT_A_UNICODE_START + NCHO_MODERN + NJONG_ANCIENT;
00815 else
00816     cho_num = -1;
00817
00818 if (jung >= 0x1161 && jung <= 0x11A7)
00819     jung_num = jung - JUNG_UNICODE_START;
00820 else if (jung >= JUNG_EXT_B_UNICODE_START &&
00821         jung < (JUNG_EXT_B_UNICODE_START + NJUNG_EXT_B))
00822     jung_num = jung - JUNG_EXT_B_UNICODE_START + NJUNG_MODERN + NJUNG_ANCIENT;
00823 else
00824     jung_num = -1;
00825
00826 if (jong >= 0x11A8 && jong <= 0x11FF)
00827     jong_num = jong - JONG_UNICODE_START;
00828 else if (jong >= JONG_EXT_B_UNICODE_START &&
00829         jong < (JONG_EXT_B_UNICODE_START + NJONG_EXT_B))
00830     jong_num = jong - JONG_EXT_B_UNICODE_START + NJONG_MODERN + NJONG_ANCIENT;
00831 else
00832     jong_num = -1;
00833
00834 /*
00835  * Choose initial consonant (choseong) variation based upon
00836  * the vowel (jungseong) if both are specified.
00837  */
00838 if (cho_num < 0) {
00839     cho_index = cho_group = 0; /* Use blank glyph for choseong. */
00840 }
00841 else {
00842     if (jung_num < 0 && jong_num < 0) { /* Choseong is by itself. */
00843         cho_group = 0;
00844         if (cho_index < (NCHO_MODERN + NCHO_ANCIENT))
00845             cho_index = cho_num + JAMO_HEX;
00846         else /* Choseong is in Hangul Jamo Extended-A range. */
00847             cho_index = cho_num - (NCHO_MODERN + NCHO_ANCIENT)
00848                             + JAMO_EXT_A_HEX;
00849     }
00850     else {
00851         if (jung_num >= 0) { /* Valid jungseong with choseong. */
00852             cho_group = cho_variation (cho_num, jung_num, jong_num);
00853         }
00854         else { /* Invalid vowel; see if final consonant is valid. */
00855             /*
00856              * If initial consonant and final consonant are specified,
00857              * set cho_group to 4, which is the group that would apply
00858              * to a horizontal-only vowel such as Hangul "O", so the
00859              * consonant appears full-width.
00860              */
00861             cho_group = 0;
00862             if (jong_num >= 0) {
00863                 cho_group = 4;
00864             }
00865         }
00866         cho_index = CHO_HEX + CHO_VARIATIONS * cho_num +
00867                     cho_group;
00868     }
00869 } /* Choseong combined with jungseong and/or jongseong. */
00870 } /* Valid choseong. */
00871
00872 /*
00873  * Choose vowel (jungseong) variation based upon the choseong

```

```

00874     and jungseong.
00875 */
00876 jung_index = jung_group = 0; /* Use blank glyph for jungseong. */
00877
00878 if (jung_num >= 0) {
00879     if (cho_num < 0 && jong_num < 0) { /* Jungseong is by itself. */
00880         jung_group = 0;
00881         jung_index = jung_num + JUNG_UNICODE_START;
00882     }
00883     else {
00884         if (jong_num >= 0) { /* If there is a final consonant. */
00885             if (jong_num == 3) /* Nieun; choose variation 3. */
00886                 jung_group = 2;
00887             else
00888                 jung_group = 1;
00889         } /* Valid jongseong. */
00890         /* If valid choseong but no jongseong, choose jungseong variation 0. */
00891         else if (cho_num >= 0)
00892             jung_group = 0;
00893     }
00894     jung_index = JUNG_HEX + JUNG_VARIATIONS * jung_num + jung_group;
00895 }
00896
00897 /*
00898  Choose final consonant (jongseong) based upon whether choseong
00899  and/or jungseong are present.
00900 */
00901 if (jong_num < 0) {
00902     jong_index = jong_group = 0; /* Use blank glyph for jongseong. */
00903 }
00904 else { /* Valid jongseong. */
00905     if (cho_num < 0 && jung_num < 0) { /* Jungseong is by itself. */
00906         jong_group = 0;
00907         jong_index = jung_num + 0x4A8;
00908     }
00909     else { /* There is only one jongseong variation if combined. */
00910         jong_group = 0;
00911         jong_index = JONG_HEX + JONG_VARIATIONS * jong_num +
00912             jong_group;
00913     }
00914 }
00915
00916 /*
00917  Now that we know the index locations for choseong, jungseong, and
00918  jongseong glyphs, combine them into one glyph.
00919 */
00920 combine_glyphs (glyph_table [cho_index], glyph_table [jung_index],
00921               combined_glyph);
00922
00923 if (jong_index > 0) {
00924     /*
00925      If the vowel has a vertical stroke that is one column
00926      away from the right border, shift this jongseung right
00927      by one column to line up with the rightmost vertical
00928      stroke in the vowel.
00929     */
00930     if (is_wide_vowel (jung_num)) {
00931         for (i = 0; i < 16; i++) {
00932             tmp_glyph [i] = glyph_table [jong_index] [i] » 1;
00933         }
00934         combine_glyphs (combined_glyph, tmp_glyph,
00935                       combined_glyph);
00936     }
00937     else {
00938         combine_glyphs (combined_glyph, glyph_table [jong_index],
00939                       combined_glyph);
00940     }
00941 }
00942
00943 return;
00944 }

```

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;
00262     *jung_index = JUNG_HEX + jungseong * JUNG_VARIATIONS + jung_variation;;
00263     *jong_index = jongseong < 0 ? 0x0000 :
00264         JONG_HEX + jongseong * JONG_VARIATIONS + jong_variation;
00265
00266     return;
00267 }
```

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 diphthongs (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 diphthongs (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 (inststring, MAXLINE, infp) != NULL) {
00073         sscanf (inststring, "%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 (&inststring[i], "%2hhX", &base[codept][j]);
00083                     i += 2;
00084                 }
00085                 if (codept > max_codept) max_codept = codept;
00086             }
00087         }
00088     }
00089     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.

Parameters

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

Returns

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

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

```

00584 {
00585
00586     int    i; /* loop variable */
00587     int    cho_hex, jung_hex, jong_hex;
00588     unsigned char glyph_byte;
00589
00590
00591     hangul_hex_indices (choseong, jungseong, jongseong,
00592                        &cho_hex, &jung_hex, &jong_hex);
00593
00594     for (i = 0; i < 32; i++) {
00595         glyph_byte = hangul_base [cho_hex][i];
00596         glyph_byte |= hangul_base [jung_hex][i];
00597         if (jong_hex >= 0) glyph_byte |= hangul_base [jong_hex][i];
00598         syllable[i] = glyph_byte;
00599     }
00600
00601     return;
00602 }
```

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     /*
00439      Modern Jungseong in positions 0..20.
00440     */
00441     /* Location Variations Unicode Range Vowel # Vowel Names */
00442     /* ----- */
00443     /* 0x2FB */ 0, 1, 0, // U+1161..U+1163-->[ 0.. 2] A, AE, YA
00444     /* 0x304 */ 1, 0, 1, // U+1164..U+1166-->[ 3.. 5] YAE, EO, E
00445     /* 0x30D */ 0, 1, // U+1167..U+1168-->[ 6.. 7] YEO, YE
00446     /* 0x313 */ 0, // U+1169 -->[ 8] O
00447     /* 0x316 */ 0, 1, 0, // U+116A..U+116C-->[ 9..11] WA, WAE, WE
00448     /* 0x31F */ 0, 0, // U+116D..U+116E-->[12..13] YO, U
00449     /* 0x325 */ 0, 1, 0, // U+116F..U+1171-->[14..16] WEO, WE, WI
00450     /* 0x32E */ 0, 0, // U+1172..U+1173-->[17..18] YU, EU
00451     /* 0x334 */ 0, // U+1174 -->[19] YI
00452     /* 0x337 */ 0, // U+1175 -->[20] I
00453     /*
00454      Ancient Jungseong in positions 21..70.
00455     */
00456     /* Location Variations Unicode Range Vowel # Vowel Names */
00457     /* ----- */
00458     /* 0x33A */ 0, 0, 0, // U+1176..U+1178-->[21..23] A-O, A-U, YA-O
00459     /* 0x343 */ 0, 0, 0, // U+1179..U+117B-->[24..26] YA-YO, EO-O, EU-U
00460     /* 0x34C */ 0, 0, 0, // U+117C..U+117E-->[27..29] EO-EU, YEO-O, YEO-U
00461     /* 0x355 */ 0, 1, 1, // U+117F..U+1181-->[30..32] O-EO, O-E, O-YE,
00462     /* 0x35E */ 0, 0, 0, // U+1182..U+1184-->[33..35] O-O, O-U, YO-YA,
00463     /* 0x367 */ 1, 0, 0, // U+1185..U+1187-->[36..38] YO-YAE, YO-YEO, YO-O,
00464     /* 0x370 */ 0, 0, 1, // U+1188..U+118A-->[39..41] YO-I, U-A, U-AE,
00465     /* 0x379 */ 0, 1, 0, // U+118B..U+118D-->[42..44] U-EO-EU, U-YE, U-U,
00466     /* 0x382 */ 0, 0, 1, // U+118E..U+1190-->[45..47] YU-A, YU-EO, YU-E,
00467     /* 0x38B */ 0, 1, 0, // U+1191..U+1193-->[48..50] YU-YEO, YU-YE, YU-U,
00468     /* 0x394 */ 0, 0, 0, // U+1194..U+1196-->[51..53] YU-I, EU-U, EU-EU,
00469     /* 0x39D */ 0, 0, 0, // U+1197..U+1199-->[54..56] YI-U, I-A, I-YA,
00470     /* 0x3A6 */ 0, 0, 0, // U+119A..U+119C-->[57..59] I-O, I-U, I-EU,
00471     /* 0x3AF */ 0, 0, 0, // U+119D..U+119F-->[60..62] I-ARAE, ARAEA, ARAEA-EO,
00472     /* 0x3B8 */ 0, 0, 0, // U+11A0..U+11A2-->[63..65] ARAEA-U, ARAEA-I, SSANGARAE,
```

```

00473 /* 0x3C1: */ 0, 0, 0, // U+11A3..U+11A5-->[66..68] A-EU, YA-U, YEO-YA,
00474 /* 0x3CA: */ 0, 1, // U+11A6..U+11A7-->[69..70] O-YA, O-YAE
00475 #ifdef EXTENDED_HANGUL
00476 /* 0x3D0: */ 0, 0, 0, // U+D7B0..U+D7B2-->[71..73] O-YEO, O-O-I, YO-A,
00477 /* 0x3D9: */ 1, 0, 0, // U+D7B3..U+D7B5-->[74..76] YO-AE, YO-EO, U-YEO,
00478 /* 0x3E2: */ 1, 1, 0, // U+D7B6..U+D7B8-->[77..79] U-I-I, YU-AE, YU-O,
00479 /* 0x3EB: */ 0, 0, 1, // U+D7B9..U+D7BB-->[80..82] EU-A, EU-EO, EU-E,
00480 /* 0x3F4: */ 0, 0, 1, // U+D7BC..U+D7BE-->[83..85] EU-O, I-YA-O, I-YAE,
00481 /* 0x3FD: */ 0, 1, 0, // U+D7BF..U+D7C1-->[86..88] I-YEO, I-YE, I-O-I,
00482 /* 0x406: */ 0, 0, 1, // U+D7C2..U+D7C4-->[89..91] I-YO, I-YU, I-I,
00483 /* 0x40F: */ 0, 1, // U+D7C5..U+D7C6-->[92..93] ARAEA-A, ARAEA-E,
00484 /* 0x415: */ -1 // Mark end of list of vowels.
00485 #else
00486 /* 0x310: */ -1 // Mark end of list of vowels.
00487 #endif
00488 };
00489
00490
00491 if (vowel >= 0 && vowel < TOTAL_JUNG) {
00492     retval = wide_vowel [vowel];
00493 }
00494 else {
00495     retval = 0;
00496 }
00497
00498
00499 return retval;
00500 }

```

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_EXT_A_HEX;
00730 }
00731 else if (jamo >= 0xD7B0 && jamo <= 0xD7FF) {
00732     glyph_index = jamo - 0x1100 + JAMO_EXT_B_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     fprintf (fp, "%04X:", codept);
00697
00698     /* for each this_glyph row */
00699     for (i = 0; i < 16; i++) {
00700         fprintf (fp, "%04X", this_glyph[i]);
00701     }
00702     fputc ('\n', fp);
00703
00704     return;
00705 }
00706 }

```

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.](#)

```

00001 /**
00002     @file unihangul-support.c
00003
00004     @brief Functions for converting Hangul letters into syllables
00005
00006     This file contains functions for reading in Hangul letters
00007     arranged in a Johab 6/3/1 pattern and composing syllables
00008     with them. One function maps an initial letter (choseong),
00009     medial letter (jungseong), and final letter (jongseong)
00010     into the Hangul Syllables Unicode block, U+AC00..U+D7A3.
00011     Other functions allow formation of glyphs that include
00012     the ancient Hangul letters that Hanterm supported. More
00013     can be added if desired, with appropriate changes to
00014     start positions and lengths defined in "hangul.h".
00015
00016     @author Paul Hardy
00017
00018     @copyright Copyright © 2023 Paul Hardy
00019 */
00020 /*
00021     LICENSE:
00022
00023     This program is free software: you can redistribute it and/or modify
00024     it under the terms of the GNU General Public License as published by
00025     the Free Software Foundation, either version 2 of the License, or
00026     (at your option) any later version.
00027
00028     This program is distributed in the hope that it will be useful,
00029     but WITHOUT ANY WARRANTY; without even the implied warranty of
00030     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00031     GNU General Public License for more details.
00032
00033     You should have received a copy of the GNU General Public License
00034     along with this program. If not, see <http://www.gnu.org/licenses/>.
00035 */
00036
00037 #include <stdio.h>
00038 #include "hangul.h"
00039
00040
00041 /**
00042     @brief Read hangul-base.hex file into a unsigned char array.
00043
00044     Read a Hangul base .hex file with separate choseong, jungseong,
00045     and jongseong glyphs for syllable formation. The order is:
00046

```



```

00047     - Empty glyph in 0x0000 position.
00048     - Initial consonants (choseong).
00049     - Medial vowels and diphthongs (jungseong).
00050     - Final consonants (jongseong).
00051     - Individual letter forms in isolation, not for syllable formation.
00052
00053 The letters are arranged with all variations for one letter
00054 before continuing to the next letter. In the current
00055 encoding, there are 6 variations of choseong, 3 of jungseong,
00056 and 1 of jongseong per letter.
00057
00058 @param[in] Input file pointer; can be stdin.
00059 @param[out] Array of bit patterns, with 32 8-bit values per letter.
00060 @return The maximum code point value read in the file.
00061 */
00062 unsigned
00063 hangul_read_base8 (FILE *infp, unsigned char base[][32]) {
00064     unsigned codept;
00065     unsigned max_codept;
00066     int i, j;
00067     char instring[MAXLINE];
00068
00069     max_codept = 0;
00070
00071     while (fgets (instring, MAXLINE, infp) != NULL) {
00072         sscanf (instring, "%X", &codept);
00073         codept -= PUA_START;
00074         /* If code point is within range, add it */
00075         if (codept < MAX_GLYPHS) {
00076             /* Find the start of the glyph bitmap. */
00077             for (i = 1; instring[i] != '\0' && instring[i] != ':'; i++);
00078             if (instring[i] == ':') {
00079                 i++; /* Skip over ':' to get to start of bitmap. */
00080                 for (j = 0; j < 32; j++) {
00081                     sscanf (&instring[i], "%2hhX", &base[codept][j]);
00082                     i += 2;
00083                 }
00084                 if (codept > max_codept) max_codept = codept;
00085             }
00086         }
00087     }
00088 }
00089
00090 return max_codept;
00091 }
00092
00093 /**
00094 @brief Read hangul-base.hex file into a unsigned array.
00095
00096 Read a Hangul base .hex file with separate choseong, jungseong,
00097 and jongseong glyphs for syllable formation. The order is:
00098
00099     - Empty glyph in 0x0000 position.
00100     - Initial consonants (choseong).
00101     - Medial vowels and diphthongs (jungseong).
00102     - Final consonants (jongseong).
00103     - Individual letter forms in isolation, not for syllable formation.
00104
00105 The letters are arranged with all variations for one letter
00106 before continuing to the next letter. In the current
00107 encoding, there are 6 variations of choseong, 3 of jungseong,
00108 and 1 of jongseong per letter.
00109
00110 @param[in] Input file pointer; can be stdin.
00111 @param[out] Array of bit patterns, with 16 16-bit values per letter.
00112 @return The maximum code point value read in the file.
00113 */
00114 unsigned
00115 hangul_read_base16 (FILE *infp, unsigned base[][16]) {
00116     unsigned codept;
00117     unsigned max_codept;
00118     int i, j;
00119     char instring[MAXLINE];
00120
00121     max_codept = 0;
00122
00123     while (fgets (instring, MAXLINE, infp) != NULL) {
00124         sscanf (instring, "%X", &codept);
00125         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 }
00145
00146 /**
00147  @brief Decompose a Hangul Syllables code point into three letters.
00148
00149  Decompose a Hangul Syllables code point (U+AC00..U+D7A3) into:
00150
00151      - Choseong    0-19
00152      - Jungseong   0-20
00153      - Jongseong   0-27 or -1 if no jongseong
00154
00155  All letter values are set to -1 if the letters do not
00156  form a syllable in the Hangul Syllables range. This function
00157  only handles modern Hangul, because that is all that is in
00158  the Hangul Syllables range.
00159
00160  @param[in] codept The Unicode code point to decode, from 0xAC00 to 0xD7A3.
00161  @param[out] initial The 1st letter (choseong) in the syllable.
00162  @param[out] medial The 2nd letter (jungseong) in the syllable.
00163  @param[out] final The 3rd letter (jongseong) in the syllable.
00164  */
00165 void
00166 hangul_decompose (unsigned codept, int *initial, int *medial, int *final) {
00167     if (codept < 0xAC00 || codept > 0xD7A3) {
00168         *initial = *medial = *final = -1;
00169     }
00170     else {
00171         codept -= 0xAC00;
00172         *initial = codept / (28 * 21);
00173         *medial = (codept / 28) % 21;
00174         *final = codept % 28 - 1;
00175     }
00176     return;
00177 }
00178
00179 /**
00180  @brief Compose a Hangul syllable into a code point, or 0 if none exists.
00181
00182  This function takes three letters that can form a modern Hangul
00183  syllable and returns the corresponding Unicode Hangul Syllables
00184  code point in the range 0xAC00 to 0xD7A3.
00185
00186  If a three-letter combination includes one or more archaic letters,
00187  it will not map into the Hangul Syllables range. In that case,
00188  the returned code point will be 0 to indicate that no valid
00189  Hangul Syllables code point exists.
00190
00191  @param[in] initial The first letter (choseong), 0 to 18.
00192  @param[in] medial The second letter (jungseong), 0 to 20.
00193  @param[in] final The third letter (jongseong), 0 to 26 or -1 if none.
00194  @return The Unicode Hangul Syllables code point, 0xAC00 to 0xD7A3.
00195  */
00196 unsigned
00197 hangul_compose (int initial, int medial, int final) {
00198     if (initial >= 0 && initial <= 18 &&
00199         medial >= 0 && medial <= 20 &&
00200         final >= 0 && final <= 26) {

```

```

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 }
00220
00221
00222 /**
00223  @brief Determine index values to the bitmaps for a syllable's components.
00224
00225  This function reads these input values for modern and ancient Hangul letters:
00226
00227      - Choseong number (0 to the number of modern and archaic choseong - 1.
00228      - Jungseong number (0 to the number of modern and archaic jungseong - 1.
00229      - Jongseong number (0 to the number of modern and archaic jongseong - 1, or -1 if none.
00230
00231  It then determines the variation of each letter given the combination with
00232  the other two letters (or just choseong and jungseong if the jongseong value
00233  is -1).
00234
00235  These variations are then converted into index locations within the
00236  glyph array that was read in from the hangul-base.hex file. Those
00237  index locations can then be used to form a composite syllable.
00238
00239  There is no restriction to only use the modern Hangul letters.
00240
00241  @param[in] choseong The 1st letter in the syllable.
00242  @param[in] jungseong The 2nd letter in the syllable.
00243  @param[in] jongseong The 3rd letter in the syllable, or -1 if none.
00244  @param[out] cho_index Index location to the 1st letter variation from the hangul-base.hex file.
00245  @param[out] jung_index Index location to the 2nd letter variation from the hangul-base.hex file.
00246  @param[out] jong_index Index location to the 3rd letter variation from the hangul-base.hex file.
00247 */
00248 void
00249 hangul_hex_indices (int choseong, int jungseong, int jongseong,
00250                    int *cho_index, int *jung_index, int *jong_index) {
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     hangul_variations (choseong, jungseong, jongseong,
00258                      &cho_variation, &jung_variation, &jong_variation);
00259
00260     *cho_index = CHO_HEX + choseong * CHO_VARIATIONS + cho_variation;
00261     *jung_index = JUNG_HEX + jungseong * JUNG_VARIATIONS + jung_variation;
00262     *jong_index = jongseong < 0 ? 0x0000 :
00263                   JONG_HEX + jongseong * JONG_VARIATIONS + jong_variation;
00264
00265     return;
00266 }
00267
00268
00269
00270 /**
00271  @brief Determine the variations of each letter in a Hangul syllable.
00272
00273  Given the three letters that will form a syllable, return the variation
00274  of each letter used to form the composite glyph.
00275
00276  This function can determine variations for both modern and archaic
00277  Hangul letters; it is not limited to only the letters combinations
00278  that comprise the Unicode Hangul Syllables range.
00279
00280  This function reads these input values for modern and ancient Hangul letters:
00281
00282      - Choseong number (0 to the number of modern and archaic choseong - 1.
00283      - Jungseong number (0 to the number of modern and archaic jungseong - 1.
00284      - Jongseong number (0 to the number of modern and archaic jongseong - 1, or -1 if none.
00285
00286  It then determines the variation of each letter given the combination with
00287  the other two letters (or just choseong and jungseong if the jongseong value
00288  is -1).
00289

```

```

00290  @param[in] choseong The 1st letter in the syllable.
00291  @param[in] jungseong The 2nd letter in the syllable.
00292  @param[in] jongseong The 3rd letter in the syllable, or -1 if none.
00293  @param[out] cho_var Variation of the 1st letter from the hangul-base.hex file.
00294  @param[out] jung_var Variation of the 2nd letter from the hangul-base.hex file.
00295  @param[out] jong_var Variation of the 3rd letter from the hangul-base.hex file.
00296  */
00297  void
00298  hangul_variations (int choseong, int jungseong, int jongseong,
00299                    int *cho_var, int *jung_var, int *jong_var) {
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  }
00315
00316
00317  /**
00318   @brief Return the Johab 6/3/1 choseong variation for a syllable.
00319
00320   This function takes the two or three (if jongseong is included)
00321   letters that comprise a syllable and determine the variation
00322   of the initial consonant (choseong).
00323
00324   Each choseong has 6 variations:
00325
00326       Variation  Occurrence
00327       -----
00328           0      Choseong with a vertical vowel such as "A".
00329           1      Choseong with a horizontal vowel such as "O".
00330           2      Choseong with a vertical and horizontal vowel such as "WA".
00331           3      Same as variation 0, but with jongseong (final consonant).
00332           4      Same as variation 1, but with jongseong (final consonant).
00333                   Also a horizontal vowel pointing down, such as U and YU.
00334           5      Same as variation 2, but with jongseong (final consonant).
00335                   Also a horizontal vowel pointing down with vertical element,
00336                   such as WEO, WE, and WI.
00337
00338   In addition, if the vowel is horizontal and a downward-pointing stroke
00339   as in the modern letters U, WEO, WE, WI, and YU, and in archaic
00340   letters YU-YEO, YU-YE, YU-I, araea, and araea-i, then 3 is added
00341   to the initial variation of 0 to 2, resulting in a choseong variation
00342   of 3 to 5, respectively.
00343
00344   @param[in] choseong The 1st letter in the syllable.
00345   @param[in] jungseong The 2nd letter in the syllable.
00346   @param[in] jongseong The 3rd letter in the syllable.
00347   @return The choseong variation, 0 to 5.
00348  */
00349  int
00350  cho_variation (int choseong, int jungseong, int jongseong) {
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 #ifndef 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) {
00414     cho_variation = -1;
00415 }
00416 else {
00417     cho_variation = choseong_var [jungseong];
00418     if (choseong >= 0 && jongseong >= 0 && cho_variation < 3)
00419         cho_variation += 3;
00420 }
00421
00422
00423 return cho_variation;
00424 }
00425
00426
00427 /**
00428  @brief Whether vowel has rightmost vertical stroke to the right.
00429
00430  @param[in] vowel Vowel number, from 0 to TOTAL_JUNG - 1.
00431  @return 1 if this vowel's vertical stroke is wide on the right side; else 0.
00432 */
00433 int
00434 is_wide_vowel (int vowel) {
00435     int retval; /* Return value. */
00436
00437     static int wide_vowel [TOTAL_JUNG + 1] = {
00438         /*
00439             Modern Jungseong in positions 0..20.
00440         */
00441         /* Location Variations Unicode Range Vowel # Vowel Names */
00442         /* ----- */
00443         /* 0x2FB */ 0, 1, 0, // U+1161..U+1163-->[ 0.. 2] A, AE, YA
00444         /* 0x304 */ 1, 0, 1, // U+1164..U+1166-->[ 3.. 5] YAE, EO, E
00445         /* 0x30D */ 0, 1, // U+1167..U+1168-->[ 6.. 7] YEO, YE
00446         /* 0x313 */ 0, // U+1169 -->[ 8] O
00447         /* 0x316 */ 0, 1, 0, // U+116A..U+116C-->[ 9..11] WA, WAE, WE
00448         /* 0x31F */ 0, 0, // U+116D..U+116E-->[12..13] YO, U
00449         /* 0x325 */ 0, 1, 0, // U+116F..U+1171-->[14..16] WEO, WE, WI
00450         /* 0x32E */ 0, 0, // U+1172..U+1173-->[17..18] YU, EU
00451         /* 0x334 */ 0, // U+1174 -->[19] YI

```

```

00452 /* 0x337 */ 0, // U+1175 -->[20] I
00453 /*
00454     * Ancient Jungseong in positions 21..70.
00455     */
00456 /* Location Variations Unicode Range Vowel # Vowel Names */
00457 /* ----- */
00458 /* 0x33A: */ 0, 0, 0, // U+1176..U+1178-->[21..23] A-O, A-U, YA-O
00459 /* 0x343: */ 0, 0, 0, // U+1179..U+117B-->[24..26] YA-YO, EO-O, EU-U
00460 /* 0x34C: */ 0, 0, 0, // U+117C..U+117E-->[27..29] EO-EU, YEO-O, YEO-U
00461 /* 0x355: */ 0, 1, 1, // U+117F..U+1181-->[30..32] O-EO, O-E, O-YE,
00462 /* 0x35E: */ 0, 0, 0, // U+1182..U+1184-->[33..35] O-O, O-U, YO-YA,
00463 /* 0x367: */ 1, 0, 0, // U+1185..U+1187-->[36..38] YO-YAE, YO-YEO, YO-O,
00464 /* 0x370: */ 0, 0, 1, // U+1188..U+118A-->[39..41] YO-I, U-A, U-AE,
00465 /* 0x379: */ 0, 1, 0, // U+118B..U+118D-->[42..44] U-EO-EU, U-YE, U-U,
00466 /* 0x382: */ 0, 0, 1, // U+118E..U+1190-->[45..47] YU-A, YU-EO, YU-E,
00467 /* 0x38B: */ 0, 1, 0, // U+1191..U+1193-->[48..50] YU-YEO, YU-YE, YU-U,
00468 /* 0x394: */ 0, 0, 0, // U+1194..U+1196-->[51..53] YU-I, EU-U, EU-EU,
00469 /* 0x39D: */ 0, 0, 0, // U+1197..U+1199-->[54..56] YI-U, I-A, I-YA,
00470 /* 0x3A6: */ 0, 0, 0, // U+119A..U+119C-->[57..59] I-O, I-U, I-EU,
00471 /* 0x3AF: */ 0, 0, 0, // U+119D..U+119F-->[60..62] I-ARAEA, ARAEA, ARAEA-EO,
00472 /* 0x3B8: */ 1, 0, 0, // U+11A0..U+11A2-->[63..65] ARAEA-U, ARAEA-I, SSANGARAEA,
00473 /* 0x3C1: */ 0, 0, 0, // U+11A3..U+11A5-->[66..68] A-EU, YA-U, YEO-YA,
00474 /* 0x3CA: */ 0, 1, // U+11A6..U+11A7-->[69..70] O-YA, O-YAE
00475 #ifdef EXTENDED_HANGUL
00476 /* 0x3D0: */ 0, 0, 0, // U+D7B0..U+D7B2-->[71..73] O-YEO, O-O-I, YO-A,
00477 /* 0x3D9: */ 1, 0, 0, // U+D7B3..U+D7B5-->[74..76] YO-AE, YO-EO, U-YEO,
00478 /* 0x3E2: */ 1, 1, 0, // U+D7B6..U+D7B8-->[77..79] U-I-I, YU-AE, YU-O,
00479 /* 0x3EB: */ 0, 0, 1, // U+D7B9..U+D7BB-->[80..82] EU-A, EU-EO, EU-E,
00480 /* 0x3F4: */ 0, 0, 1, // U+D7BC..U+D7BE-->[83..85] EU-O, I-YA-O, I-YAE,
00481 /* 0x3FD: */ 0, 1, 0, // U+D7BF..U+D7C1-->[86..88] I-YEO, I-YE, I-O-I,
00482 /* 0x406: */ 0, 0, 1, // U+D7C2..U+D7C4-->[89..91] I-YO, I-YU, I-I,
00483 /* 0x40F: */ 0, 1, // U+D7C5..U+D7C6-->[92..93] ARAEA-A, ARAEA-E,
00484 /* 0x415: */ -1 // Mark end of list of vowels.
00485 #else
00486 /* 0x310: */ -1 // Mark end of list of vowels.
00487 #endif
00488 };
00489
00490
00491 if (vowel >= 0 && vowel < TOTAL_JUNG) {
00492     retval = wide_vowel[vowel];
00493 }
00494 else {
00495     retval = 0;
00496 }
00497
00498
00499 return retval;
00500 }
00501
00502
00503 /**
00504     @brief Return the Johab 6/3/1 jungseong variation.
00505
00506     This function takes the two or three (if jongseong is included)
00507     letters that comprise a syllable and determine the variation
00508     of the vowel (jungseong).
00509
00510     Each jungseong has 3 variations:
00511
00512     Variation Occurrence
00513     -----
00514     0 Jungseong with only chungseong (no jongseong).
00515     1 Jungseong with chungseong and jungseong (except nieun).
00516     2 Jungseong with chungseong and jungseong nieun.
00517
00518     @param[in] choseong The 1st letter in the syllable.
00519     @param[in] jungseong The 2nd letter in the syllable.
00520     @param[in] jongseong The 3rd letter in the syllable.
00521     @return The jungseong variation, 0 to 2.
00522 */
00523 inline int
00524 jung_variation (int choseong, int jungseong, int jongseong) {
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 }
00543
00544
00545 /**
00546  @brief Return the Johab 6/3/1 jongseong variation.
00547
00548  There is only one jongseong variation, so this function
00549  always returns 0. It is a placeholder function for
00550  possible future adaptation to other johab encodings.
00551
00552  @param[in] choseong The 1st letter in the syllable.
00553  @param[in] jungseong The 2nd letter in the syllable.
00554  @param[in] jongseong The 3rd letter in the syllable.
00555  @return The jongseong variation, always 0.
00556 */
00557 inline int
00558 jung_variation (int choseong, int jungseong, int jongseong) {
00559
00560     return 0; /* There is only one Jongseong variation. */
00561 }
00562
00563
00564 /**
00565  @brief Given letters in a Hangul syllable, return a glyph.
00566
00567  This function returns a glyph bitmap comprising up to three
00568  Hangul letters that form a syllable. It reads the three
00569  component letters (choseong, jungseong, and jongseong),
00570  then calls a function that determines the appropriate
00571  variation of each letter, returning the letter bitmap locations
00572  in the glyph array. Then these letter bitmaps are combined
00573  with a logical OR operation to produce a final bitmap,
00574  which forms a 16 row by 16 column bitmap glyph.
00575
00576  @param[in] choseong The 1st letter in the composite glyph.
00577  @param[in] jungseong The 2nd letter in the composite glyph.
00578  @param[in] jongseong The 3rd letter in the composite glyph.
00579  @param[in] hangul_base The glyphs read from the "hangul_base.hex" file.
00580  @return syllable The composite syllable, as a 16 by 16 pixel bitmap.
00581 */
00582 void
00583 hangul_syllable (int choseong, int jungseong, int jongseong,
00584                 unsigned char hangul_base[][32], unsigned char *syllable) {
00585
00586     int i; /* loop variable */
00587     int cho_hex, jung_hex, jong_hex;
00588     unsigned char glyph_byte;
00589
00590
00591     hangul_hex_indices (choseong, jungseong, jongseong,
00592                        &cho_hex, &jung_hex, &jong_hex);
00593
00594     for (i = 0; i < 32; i++) {
00595         glyph_byte = hangul_base [cho_hex][i];
00596         glyph_byte |= hangul_base [jung_hex][i];
00597         if (jong_hex >= 0) glyph_byte |= hangul_base [jong_hex][i];
00598         syllable[i] = glyph_byte;
00599     }
00600
00601     return;
00602 }
00603
00604
00605 /**
00606  @brief See if two glyphs overlap.
00607
00608  @param[in] glyph1 The first glyph, as a 16-row bitmap.
00609  @param[in] glyph2 The second glyph, as a 16-row bitmap.
00610  @return 0 if no overlaps between glyphs, 1 otherwise.
00611 */
00612 int
00613 glyph_overlap (unsigned *glyph1, unsigned *glyph2) {

```

```

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 }
00627
00628
00629 /**
00630  @brief Combine two glyphs into one glyph.
00631
00632  @param[in] glyph1 The first glyph to overlap.
00633  @param[in] glyph2 The second glyph to overlap.
00634  @param[out] combined_glyph The returned combination glyph.
00635 */
00636 void
00637 combine_glyphs (unsigned *glyph1, unsigned *glyph2,
00638               unsigned *combined_glyph) {
00639     int i;
00640
00641     for (i = 0; i < 16; i++)
00642         combined_glyph[i] = glyph1[i] | glyph2[i];
00643
00644     return;
00645 }
00646
00647
00648 /**
00649  @brief Print one glyph in Unifont hexdraw plain text style.
00650
00651  @param[in] fp      The file pointer for output.
00652  @param[in] codept  The Unicode code point to print with the glyph.
00653  @param[in] this_glyph The 16-row by 16-column glyph to print.
00654 */
00655 void
00656 print_glyph_txt (FILE *fp, unsigned codept, unsigned *this_glyph) {
00657     int i;
00658     unsigned mask;
00659
00660     fprintf (fp, "%04X:", codept);
00661
00662     /* for each this_glyph row */
00663     for (i = 0; i < 16; i++) {
00664         mask = 0x8000;
00665         fputc ('\t', fp);
00666         while (mask != 0x0000) {
00667             if (mask & this_glyph[i]) {
00668                 fputc ('#', fp);
00669             }
00670             else {
00671                 fputc ('.', fp);
00672             }
00673             mask »= 1; /* shift to next bit in this_glyph row */
00674         }
00675         fputc ('\n', fp);
00676     }
00677     fputc ('\n', fp);
00678
00679     return;
00680 }
00681
00682
00683
00684 /**
00685  @brief Print one glyph in Unifont hexdraw hexadecimal string style.
00686
00687  @param[in] fp      The file pointer for output.
00688  @param[in] codept  The Unicode code point to print with the glyph.
00689  @param[in] this_glyph The 16-row by 16-column glyph to print.
00690 */
00691 void
00692 print_glyph_hex (FILE *fp, unsigned codept, unsigned *this_glyph) {
00693     int i;

```



```

00695
00696
00697     fprintf (fp, "%04X:", codept);
00698
00699     /* for each this_glyph row */
00700     for (i = 0; i < 16; i++) {
00701         fprintf (fp, "%04X", this_glyph[i]);
00702     }
00703     fputc ('\n', fp);
00704
00705     return;
00706 }
00707
00708
00709 /**
00710  @brief Convert Hangul Jamo choseong, jungseong, and jongseong into a glyph.
00711
00712  @param[in]  glyph_table The collection of all jamo glyphs.
00713  @param[in]  jamo        The Unicode code point, 0 or 0x1100..0x115F.
00714  @param[out] jamo_glyph  The output glyph, 16 columns in each of 16 rows.
00715  */
00716 void
00717 one_jamo (unsigned glyph_table [MAX_GLYPHS][16],
00718          unsigned jamo, unsigned *jamo_glyph) {
00719
00720     int i; /* Loop variable */
00721     int glyph_index; /* Location of glyph in "hangul-base.hex" array */
00722
00723     /* If jamo is invalid range, use blank glyph, */
00724     if (jamo >= 0x1100 && jamo <= 0x11FF) {
00725         glyph_index = jamo - 0x1100 + JAMO_HEX;
00726     }
00727     else if (jamo >= 0xA960 && jamo <= 0xA97F) {
00728         glyph_index = jamo - 0xA960 + JAMO_EXT_A_HEX;
00729     }
00730     else if (jamo >= 0xD7B0 && jamo <= 0xD7FF) {
00731         glyph_index = jamo - 0x1100 + JAMO_EXT_B_HEX;
00732     }
00733     else {
00734         glyph_index = 0;
00735     }
00736
00737     for (i = 0; i < 16; i++) {
00738         jamo_glyph[i] = glyph_table[glyph_index][i];
00739     }
00740
00741     return;
00742 }
00743
00744
00745
00746 /**
00747  @brief Convert Hangul Jamo choseong, jungseong, and jongseong into a glyph.
00748
00749  This function converts input Hangul choseong, jungseong, and jongseong
00750  Unicode code triplets into a Hangul syllable. Any of those with an
00751  out of range code point are assigned a blank glyph for combining.
00752
00753  This function performs the following steps:
00754
00755  1) Determine the sequence number of choseong, jungseong,
00756     and jongseong, from 0 to the total number of choseong,
00757     jungseong, or jongseong, respectively, minus one. The
00758     sequence for each is as follows:
00759
00760     a) Choseong: Unicode code points of U+1100..U+115E
00761        and then U+A960..U+A97C.
00762
00763     b) Jungseong: Unicode code points of U+1161..U+11A7
00764        and then U+D7B0..U+D7C6.
00765
00766     c) Jongseong: Unicode code points of U+11A8..U+11FF
00767        and then U+D7CB..U+D7FB.
00768
00769  2) From the choseong, jungseong, and jongseong sequence number,
00770     determine the variation of choseong and jungseong (there is
00771     only one jongseong variation, although it is shifted right
00772     by one column for some vowels with a pair of long vertical
00773     strokes on the right side).
00774
00775  3) Convert the variation numbers for the three syllable

```

```

00776         components to index locations in the glyph array.
00777
00778     4) Combine the glyph array glyphs into a syllable.
00779
00780     @param[in] glyph_table The collection of all jamo glyphs.
00781     @param[in] cho The choseong Unicode code point, 0 or 0x1100..0x115F.
00782     @param[in] jung The jungseong Unicode code point, 0 or 0x1160..0x11A7.
00783     @param[in] jong The jongseong Unicode code point, 0 or 0x11A8..0x11FF.
00784     @param[out] combined_glyph The output glyph, 16 columns in each of 16 rows.
00785 */
00786 void
00787 combine_jamo (unsigned glyph_table [MAX_GLYPHS][16],
00788             unsigned cho, unsigned jung, unsigned jong,
00789             unsigned *combined_glyph) {
00790
00791     int i; /* Loop variable. */
00792     int cho_num, jung_num, jong_num;
00793     int cho_group, jung_group, jong_group;
00794     int cho_index, jung_index, jong_index;
00795
00796     unsigned tmp_glyph[16]; /* Hold shifted jongsung for wide vertical vowel. */
00797
00798     int cho_variation (int choseong, int jungseong, int jongseong);
00799
00800     void combine_glyphs (unsigned *glyph1, unsigned *glyph2,
00801                        unsigned *combined_glyph);
00802
00803     /* Choose a blank glyph for each syllable by default. */
00804     cho_index = jung_index = jong_index = 0x000;
00805
00806     /*
00807      * Convert Unicode code points to jamo sequence number
00808      * of each letter, or -1 if letter is not in valid range.
00809      */
00810     if (cho >= 0x1100 && cho <= 0x115E)
00811         cho_num = cho - CHO_UNICODE_START;
00812     else if (cho >= CHO_EXTB_UNICODE_START &&
00813             cho < (CHO_EXTB_UNICODE_START + NCHO_EXTB))
00814         cho_num = cho - CHO_EXTB_UNICODE_START + NCHO_MODERN + NJONG_ANCIENT;
00815     else
00816         cho_num = -1;
00817
00818     if (jung >= 0x1161 && jung <= 0x11A6)
00819         jung_num = jung - JUNG_UNICODE_START;
00820     else if (jung >= JUNG_EXTB_UNICODE_START &&
00821             jung < (JUNG_EXTB_UNICODE_START + NJUNG_EXTB))
00822         jung_num = jung - JUNG_EXTB_UNICODE_START + NJUNG_MODERN + NJUNG_ANCIENT;
00823     else
00824         jung_num = -1;
00825
00826     if (jong >= 0x11A8 && jong <= 0x11FF)
00827         jong_num = jong - JONG_UNICODE_START;
00828     else if (jong >= JONG_EXTB_UNICODE_START &&
00829             jong < (JONG_EXTB_UNICODE_START + NJONG_EXTB))
00830         jong_num = jong - JONG_EXTB_UNICODE_START + NJONG_MODERN + NJONG_ANCIENT;
00831     else
00832         jong_num = -1;
00833
00834     /*
00835      * Choose initial consonant (choseong) variation based upon
00836      * the vowel (jungseong) if both are specified.
00837      */
00838     if (cho_num < 0) {
00839         cho_index = cho_group = 0; /* Use blank glyph for choseong. */
00840     }
00841     else {
00842         if (jung_num < 0 && jong_num < 0) { /* Choseong is by itself. */
00843             cho_group = 0;
00844             if (cho_index < (NCHO_MODERN + NCHO_ANCIENT))
00845                 cho_index = cho_num + JAMO_HEX;
00846             else /* Choseong is in Hangul Jamo Extended-A range. */
00847                 cho_index = cho_num - (NCHO_MODERN + NCHO_ANCIENT)
00848                     + JAMO_EXTB_HEX;
00849         }
00850         else {
00851             if (jung_num >= 0) { /* Valid jungseong with choseong. */
00852                 cho_group = cho_variation (cho_num, jung_num, jong_num);
00853             }
00854             else { /* Invalid vowel; see if final consonant is valid. */
00855                 /*

```

```

00857         If initial consonant and final consonant are specified,
00858         set cho_group to 4, which is the group tha would apply
00859         to a horizontal-only vowel such as Hangul "O", so the
00860         consonant appears full-width.
00861         */
00862         cho_group = 0;
00863         if (jong_num >= 0) {
00864             cho_group = 4;
00865         }
00866     }
00867     cho_index = CHO_HEX + CHO_VARIATIONS * cho_num +
00868         cho_group;
00869 } /* Choseong combined with jungseong and/or jongseong. */
00870 } /* Valid choseong. */
00871
00872 /*
00873     Choose vowel (jungseong) variation based upon the choseong
00874     and jungseong.
00875     */
00876 jung_index = jung_group = 0; /* Use blank glyph for jungseong. */
00877
00878 if (jung_num >= 0) {
00879     if (cho_num < 0 && jong_num < 0) { /* Jungseong is by itself. */
00880         jung_group = 0;
00881         jung_index = jung_num + JUNG_UNICODE_START;
00882     }
00883     else {
00884         if (jong_num >= 0) { /* If there is a final consonant. */
00885             if (jong_num == 3) /* Nieun; choose variation 3. */
00886                 jung_group = 2;
00887             else
00888                 jung_group = 1;
00889         } /* Valid jongseong. */
00890         /* If valid choseong but no jongseong, choose jungseong variation 0. */
00891         else if (cho_num >= 0)
00892             jung_group = 0;
00893     }
00894     jung_index = JUNG_HEX + JUNG_VARIATIONS * jung_num + jung_group;
00895 }
00896
00897 /*
00898     Choose final consonant (jongseong) based upon whether choseong
00899     and/or jungseong are present.
00900     */
00901 if (jong_num < 0) {
00902     jong_index = jong_group = 0; /* Use blank glyph for jongseong. */
00903 }
00904 else { /* Valid jongseong. */
00905     if (cho_num < 0 && jung_num < 0) { /* Jongseong is by itself. */
00906         jong_group = 0;
00907         jong_index = jong_num + 0x4A8;
00908     }
00909     else { /* There is only one jongseong variation if combined. */
00910         jong_group = 0;
00911         jong_index = JONG_HEX + JONG_VARIATIONS * jong_num +
00912             jong_group;
00913     }
00914 }
00915
00916 /*
00917     Now that we know the index locations for choseong, jungseong, and
00918     jongseong glyphs, combine them into one glyph.
00919     */
00920 combine_glyphs (glyph_table [cho_index], glyph_table [jung_index],
00921     combined_glyph);
00922
00923 if (jong_index > 0) {
00924     /*
00925         If the vowel has a vertical stroke that is one column
00926         away from the right border, shift this jongseung right
00927         by one column to line up with the rightmost vertical
00928         stroke in the vowel.
00929         */
00930     if (is_wide_vowel (jung_num)) {
00931         for (i = 0; i < 16; i++) {
00932             tmp_glyph [i] = glyph_table [jong_index] [i] » 1;
00933         }
00934         combine_glyphs (combined_glyph, tmp_glyph,
00935             combined_glyph);
00936     }
00937     else {

```

```

00938     combine_glyphs (combined_glyph, glyph_table [jong_index],
00939                   combined_glyph);
00940     }
00941 }
00942
00943 return;
00944 }
00945

```

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 */
00428         for (j=0; j<32; j++) hexbits[i][j] = ~charbits[j][1];
00429     }
00430 }
00431
00432 /*
00433  * Initialize bitmap to all white.
00434  */
00435 for (toppixelrow=0; toppixelrow < 17*32; toppixelrow++) {
00436     for (thiscol=0; thiscol<18; thiscol++) {
00437         bitmap[toppixelrow][(thiscol « 2) ] = 0xff;
00438         bitmap[toppixelrow][(thiscol « 2) | 1] = 0xff;
00439         bitmap[toppixelrow][(thiscol « 2) | 2] = 0xff;
00440         bitmap[toppixelrow][(thiscol « 2) | 3] = 0xff;
00441     }
00442 }
00443 }
00444 /*
00445  * Write the "u+nnnn" table header in the upper left-hand corner,
00446  * where nnnn is the upper 16 bits of a 32-bit Unicode assignment.
00447  */
00448 pnybble3 = (unipage » 20);
00449 pnybble2 = (unipage » 16) & 0xf;
00450 pnybble1 = (unipage » 12) & 0xf;
00451 pnybble0 = (unipage » 8) & 0xf;
00452 for (i=0; i<32; i++) {
00453     bitmap[i][1] = hexbits[16][i]; /* copy 'u' */
00454     bitmap[i][2] = hexbits[17][i]; /* copy '+' */
00455     bitmap[i][3] = hexbits[pnybble3][i];
00456     bitmap[i][4] = hexbits[pnybble2][i];
00457     bitmap[i][5] = hexbits[pnybble1][i];
00458     bitmap[i][6] = hexbits[pnybble0][i];
00459 }
00460 /*
00461  * Write low-order 2 bytes of Unicode number assignments, as hex labels
00462  */
00463 pnybble3 = (unipage » 4) & 0xf; /* Highest-order hex digit */
00464 pnybble2 = (unipage » 0) & 0xf; /* Next highest-order hex digit */
00465 /*
00466  * Write the column headers in bitmap[] (row headers if flipped)

```

```

00467 */
00468 toppixelrow = 32 * 17 - 1; /* maximum pixel row number */
00469 /*
00470 Label the column headers. The hexbits[][] bytes are split across two
00471 bitmap[][] entries to center a the hex digits in a column of 4 bytes.
00472 OR highest byte with 0xf0 and lowest byte with 0x0f to make outer
00473 nybbles white (0=black, 1=white).
00474 */
00475 for (i=0; i<16; i++) {
00476     for (j=0; j<32; j++) {
00477         if (flip) { /* transpose matrix */
00478             bitmap[j][((i+2) << 2) | 0] = (hexbits[nybble3][j] >> 4) | 0xf0;
00479             bitmap[j][((i+2) << 2) | 1] = (hexbits[nybble3][j] << 4) |
00480                 (hexbits[nybble2][j] >> 4);
00481             bitmap[j][((i+2) << 2) | 2] = (hexbits[nybble2][j] << 4) |
00482                 (hexbits[i][j] >> 4);
00483             bitmap[j][((i+2) << 2) | 3] = (hexbits[i][j] << 4) | 0x0f;
00484         }
00485         else {
00486             bitmap[j][((i+2) << 2) | 1] = (hexbits[i][j] >> 4) | 0xf0;
00487             bitmap[j][((i+2) << 2) | 2] = (hexbits[i][j] << 4) | 0x0f;
00488         }
00489     }
00490 }
00491 /*
00492 Now use the single hex digit column graphics to label the row headers.
00493 */
00494 for (i=0; i<16; i++) {
00495     toppixelrow = 32 * (i + 1) - 1; /* from bottom to top */
00496     for (j=0; j<32; j++) {
00497         if (!flip) { /* if not transposing matrix */
00498             bitmap[toppixelrow + j][4] = hexbits[nybble3][j];
00499             bitmap[toppixelrow + j][5] = hexbits[nybble2][j];
00500         }
00501         bitmap[toppixelrow + j][6] = hexbits[i][j];
00502     }
00503 }
00504 /*
00505 Now draw grid lines in bitmap, around characters we just copied.
00506 */
00507 /* draw vertical lines 2 pixels wide */
00508 for (i=1*32; i<17*32; i++) {
00509     if ((i & 0x1f) == 7)
00510         i++;
00511     else if ((i & 0x1f) == 14)
00512         i += 2;
00513     else if ((i & 0x1f) == 22)
00514         i++;
00515     for (j=1; j<18; j++) {
00516         bitmap[i][(j << 2) | 3] &= 0xfe;
00517     }
00518 }
00519 /* draw horizontal lines 1 pixel tall */
00520 for (i=1*32-1; i<18*32-1; i+=32) {
00521     for (j=2; j<18; j++) {
00522         bitmap[i][(j << 2) | 0] = 0x00;
00523         bitmap[i][(j << 2) | 1] = 0x81;
00524         bitmap[i][(j << 2) | 2] = 0x81;
00525         bitmap[i][(j << 2) | 3] = 0x00;
00526     }
00527 }
00528 /* fill in top left corner pixel of grid */
00529 bitmap[31][7] = 0xfe;
00530
00531 return (0);
00532 }

```

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](#).

```

00100 {
00101
00102     int i, j;                /* loop variables */
00103     unsigned k0;             /* temp Unicode char variable */
00104     unsigned swap;          /* temp variable for swapping values */
00105     char inbuf[256];         /* input buffer */
00106     unsigned filesize;      /* size of file in bytes */
00107     unsigned bitmapsize;    /* size of bitmap image in bytes */
00108     unsigned thischar;      /* the current character */
00109     unsigned char thischarbyte; /* unsigned char lowest byte of Unicode char */
00110     int thischarrow;        /* row 0..15 where this character belongs */
00111     int thiscol;            /* column 0..15 where this character belongs */
00112     int toppixelrow;        /* pixel row, 0..16*32-1 */
00113     unsigned lastpage=0;    /* the last Unicode page read in font file */
00114     int wbmp=0;             /* set to 1 if writing .wbmp format file */
00115
00116     unsigned char bitmap[17*32][18*4]; /* final bitmap */
00117     unsigned char charbits[32][4]; /* bitmap for one character, 4 bytes/row */
00118
00119     char *infile="", *outfile=""; /* names of input and output files */
00120     FILE *infp, *outfp; /* file pointers of input and output files */
00121
00122     /* initializes bitmap row/col labeling, &c. */
00123     int init (unsigned char bitmap[17*32][18*4]);
00124
00125     /* convert hex string --> bitmap */
00126     int hex2bit (char *instring, unsigned char character[32][4]);
00127
00128     bitmapsize = 17*32*18*4; /* 17 rows by 18 cols, each 4 bytes */
00129
00130     if (argc > 1) {
00131         for (i = 1; i < argc; i++) {
00132             if (argv[i][0] == '-') { /* this is an option argument */
00133                 switch (argv[i][1]) {
00134                     case 'f': /* flip (transpose) glyphs in bitmap as in standard */
00135                         flip = !flip;
00136                         break;
00137                     case 'i': /* name of input file */
00138                         infile = &argv[i][2];
00139                         break;
00140                     case 'o': /* name of output file */
00141                         outfile = &argv[i][2];
00142                         break;
00143                     case 'p': /* specify a Unicode page other than default of 0 */
00144                         sscanf (&argv[i][2], "%x", &unipage); /* Get Unicode page */
00145                         break;
00146                     case 'w': /* write a .wbmp file instead of a .bmp file */
00147                         wbmp = 1;
00148                         break;
00149                     default: /* if unrecognized option, print list and exit */
00150                         fprintf (stderr, "\nSyntax:\n\n");
00151                         fprintf (stderr, "    %s -p<Unicode_Page> ", argv[0]);
00152                         fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
00153                         fprintf (stderr, "    -w specifies .wbmp output instead of ");
00154                         fprintf (stderr, "default Windows .bmp output.\n\n");
00155                         fprintf (stderr, "    -p is followed by 1 to 6 ");
00156                         fprintf (stderr, "Unicode page hex digits ");
00157                         fprintf (stderr, "(default is Page 0).\n\n");
00158                         fprintf (stderr, "\nExample:\n\n");
00159                         fprintf (stderr, "    %s -p83 -iunifont.hex -ou83.bmp\n\n",
00160                                 argv[0]);
00161                         exit (1);
00162                 }
00163             }
00164         }
00165     }

```



```

00166  /*
00167      Make sure we can open any I/O files that were specified before
00168      doing anything else.
00169  */
00170  if (strlen (infile) > 0) {
00171      if ((infp = fopen (infile, "r")) == NULL) {
00172          fprintf (stderr, "Error: can't open %s for input.\n", infile);
00173          exit (1);
00174      }
00175  }
00176  else {
00177      infp = stdin;
00178  }
00179  if (strlen (outfile) > 0) {
00180      if ((outfp = fopen (outfile, "w")) == NULL) {
00181          fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00182          exit (1);
00183      }
00184  }
00185  else {
00186      outfp = stdout;
00187  }
00188
00189  (void)init(bitmap); /* initialize bitmap with row/column headers, etc. */
00190
00191  /*
00192      Read in the characters in the page
00193  */
00194  while (lastpage <= unipage && fgets (inbuf, MAXBUF-1, infp) != NULL) {
00195      sscanf (inbuf, "%x", &thischar);
00196      lastpage = thischar » 8; /* keep Unicode page to see if we can stop */
00197      if (lastpage == unipage) {
00198          thischarbyte = (unsigned char)(thischar & 0xff);
00199          for (k0=0; inbuf[k0] != '\0'; k0++);
00200          hex2bit (&inbuf[k0], charbits); /* convert hex string to 32*4 bitmap */
00201
00202          /*
00203              Now write character bitmap upside-down in page array, to match
00204              .bmp file order. In the .wbmp' and .bmp files, white is a '1'
00205              bit and black is a '0' bit, so complement charbits[].
00206          */
00207
00208          thiscol = (thischarbyte & 0xf) + 2; /* column number will be 1..16 */
00209          thischarrow = thischarbyte » 4; /* character row number, 0..15 */
00210          if (flip) { /* swap row and column placement */
00211              swap = thiscol;
00212              thiscol = thischarrow;
00213              thischarrow = swap;
00214              thiscol += 2; /* column index starts at 1 */
00215              thischarrow -= 2; /* row index starts at 0 */
00216          }
00217          toppixelrow = 32 * (thischarrow + 1) - 1; /* from bottom to top */
00218
00219          /*
00220              Copy the center of charbits[] because hex characters only
00221              occupy rows 8 to 23 and column byte 2 (and for 16 bit wide
00222              characters, byte 3). The charbits[] array was given 32 rows
00223              and 4 column bytes for completeness in the beginning.
00224          */
00225          for (i=8; i<24; i++) {
00226              bitmap[toppixelrow + i][(thiscol « 2) | 0] =
00227                  ~charbits[i][0] & 0xff;
00228              bitmap[toppixelrow + i][(thiscol « 2) | 1] =
00229                  ~charbits[i][1] & 0xff;
00230              bitmap[toppixelrow + i][(thiscol « 2) | 2] =
00231                  ~charbits[i][2] & 0xff;
00232              /* Only use first 31 bits; leave vertical rule in 32nd column */
00233              bitmap[toppixelrow + i][(thiscol « 2) | 3] =
00234                  ~charbits[i][3] & 0xfe;
00235          }
00236          /*
00237              Leave white space in 32nd column of rows 8, 14, 15, and 23
00238              to leave 16 pixel height upper, middle, and lower guides.
00239          */
00240          bitmap[toppixelrow + 8][(thiscol « 2) | 3] |= 1;
00241          bitmap[toppixelrow + 14][(thiscol « 2) | 3] |= 1;
00242          bitmap[toppixelrow + 15][(thiscol « 2) | 3] |= 1;
00243          bitmap[toppixelrow + 23][(thiscol « 2) | 3] |= 1;
00244      }
00245  }
00246  }

```

```

00247  /*
00248     Now write the appropriate bitmap file format, either
00249     Wireless Bitmap or Microsoft Windows bitmap.
00250  */
00251  if (wbmp) { /* Write a Wireless Bitmap .wbmp format file */
00252      /*
00253       Write WBMP header
00254      */
00255      fprintf (outfp, "%c", 0x00); /* Type of image; always 0 (monochrome) */
00256      fprintf (outfp, "%c", 0x00); /* Reserved; always 0 */
00257      fprintf (outfp, "%c%c", 0x84, 0x40); /* Width = 576 pixels */
00258      fprintf (outfp, "%c%c", 0x84, 0x20); /* Height = 544 pixels */
00259      /*
00260       Write bitmap image
00261      */
00262      for (toppixelrow=0; toppixelrow <= 17*32-1; toppixelrow++) {
00263          for (j=0; j<18; j++) {
00264              fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2)  ]);
00265              fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 1]);
00266              fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 2]);
00267              fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 3]);
00268          }
00269      }
00270  }
00271  else { /* otherwise, write a Microsoft Windows .bmp format file */
00272      /*
00273       Write the .bmp file -- start with the header, then write the bitmap
00274      */
00275
00276      /* 'B', 'M' appears at start of every .bmp file */
00277      fprintf (outfp, "%c%c", 0x42, 0x4d);
00278
00279      /* Write file size in bytes */
00280      filesize = 0x3E + bitmapsizes;
00281      fprintf (outfp, "%c", (unsigned char)((filesize >> 24) & 0xff));
00282      fprintf (outfp, "%c", (unsigned char)((filesize >> 16) & 0xff));
00283      fprintf (outfp, "%c", (unsigned char)((filesize >> 8) & 0xff));
00284      fprintf (outfp, "%c", (unsigned char)(filesize & 0xff));
00285
00286      /* Reserved - 0's */
00287      fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00288
00289      /* Offset from start of file to bitmap data */
00290      fprintf (outfp, "%c%c%c%c", 0x3E, 0x00, 0x00, 0x00);
00291
00292      /* Length of bitmap info header */
00293      fprintf (outfp, "%c%c%c%c", 0x28, 0x00, 0x00, 0x00);
00294
00295      /* Width of bitmap in pixels */
00296      fprintf (outfp, "%c%c%c%c", 0x40, 0x02, 0x00, 0x00);
00297
00298      /* Height of bitmap in pixels */
00299      fprintf (outfp, "%c%c%c%c", 0x20, 0x02, 0x00, 0x00);
00300
00301      /* Planes in bitmap (fixed at 1) */
00302      fprintf (outfp, "%c%c", 0x01, 0x00);
00303
00304      /* bits per pixel (1 = monochrome) */
00305      fprintf (outfp, "%c%c", 0x01, 0x00);
00306
00307      /* Compression (0 = none) */
00308      fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00309
00310      /* Size of bitmap data in bytes */
00311      fprintf (outfp, "%c", (unsigned char)((bitmapsizes >> 24) & 0xff));
00312      fprintf (outfp, "%c", (unsigned char)((bitmapsizes >> 16) & 0xff));
00313      fprintf (outfp, "%c", (unsigned char)((bitmapsizes >> 8) & 0xff));
00314      fprintf (outfp, "%c", (unsigned char)(bitmapsizes & 0xff));
00315
00316      /* Horizontal resolution in pixels per meter */
00317      fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);
00318
00319      /* Vertical resolution in pixels per meter */
00320      fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);
00321
00322      /* Number of colors used */
00323      fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);
00324
00325      /* Number of important colors */
00326      fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);
00327

```

```

00328      /* The color black: B=0x00, G=0x00, R=0x00, Filler=0xFF */
00329      fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00330
00331      /* The color white: B=0xFF, G=0xFF, R=0xFF, Filler=0xFF */
00332      fprintf (outfp, "%c%c%c%c", 0xFF, 0xFF, 0xFF, 0x00);
00333
00334      /*
00335       Now write the raw data bits. Data is written from the lower
00336       left-hand corner of the image to the upper right-hand corner
00337       of the image.
00338      */
00339      for (toppixelrow=17*32-1; toppixelrow >= 0; toppixelrow--) {
00340          for (j=0; j<18; j++) {
00341              fprintf (outfp, "%c", bitmap[toppixelrow][(j«2)  ]);
00342              fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 1]);
00343              fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 2]);
00344
00345              fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 3]);
00346          }
00347      }
00348  }
00349  exit (0);
00350 }

```

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:

```

= {
    "0030:0000000018244242424242424180000",
    "0031:000000000818280808080808083E0000",
    "0032:0000000003C4242020C102040407E0000",
    "0033:0000000003C4242021C020242423C0000",
    "0034:00000000040C142444447E0404040000",
    "0035:000000007E4040407C020202423C0000",
    "0036:000000001C2040407C424242423C0000",
    "0037:000000007E02020404080808080000",
    "0038:000000003C4242423C424242423C0000",
    "0039:000000003C4242423E02020204380000",
    "0041:0000000018242442427E424242420000",
    "0042:000000007C4242427C424242427C0000",
    "0043:000000003C42424040404042423C0000",
    "0044:00000000784442424242424244780000",
    "0045:000000007E4040407C404040407E0000",
    "0046:000000007E4040407C404040400000",
    "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.](#)

```
00001 /**
00002  @file unihex2bmp.c
00003
00004  @brief unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points
00005           into a bitmap for editing
00006
00007  @author Paul Hardy, unifoundry <at> unifoundry.com, December 2007
00008
00009  @copyright Copyright (C) 2007, 2008, 2013, 2017 Paul Hardy
00010
00011  This program reads in a GNU Unifont .hex file, extracts a range of
00012  256 code points, and converts it a Microsoft Bitmap (.bmp) or Wireless
00013  Bitmap file.
00014
00015  Synopsis: unihex2bmp [-iin_file.hex] [-oout_file.bmp]
00016             [-f] [-phex_page_num] [-w]
00017 */
00018 /*
00019  LICENSE:
00020
00021  This program is free software: you can redistribute it and/or modify
00022  it under the terms of the GNU General Public License as published by
00023  the Free Software Foundation, either version 2 of the License, or
00024  (at your option) any later version.
00025
00026  This program is distributed in the hope that it will be useful,
00027  but WITHOUT ANY WARRANTY; without even the implied warranty of
00028  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00029  GNU General Public License for more details.
00030
00031  You should have received a copy of the GNU General Public License
00032  along with this program. If not, see <http://www.gnu.org/licenses/>.
00033 */
00034
00035 /*
00036  20 June 2017 [Paul Hardy]:
00037  - Adds capability to output triple-width and quadruple-width (31 pixels
00038  wide, not 32) glyphs. The 32nd column in a glyph cell is occupied by
00039  the vertical cell border, so a quadruple-width glyph can only occupy
00040  the first 31 columns; the 32nd column is ignored.
00041
00042  21 October 2023 [Paul Hardy]:
00043  - Added full prototypes in main function for init and hex2bit functions.
00044 */
00045
00046 #include <stdio.h>
00047 #include <stdlib.h>
00048 #include <string.h>
00049
00050 #define MAXBUF 256
00051
00052
00053 /**
00054  @brief GNU Unifont bitmaps for hexadecimal digits.
00055
00056  These are the GNU Unifont hex strings for '0'-'9' and 'A'-'F',
00057  for encoding as bit strings in row and column headers.
00058
```

```

00059 Looking at the final bitmap as a grid of 32*32 bit tiles, the
00060 first row contains a hexadecimal character string of the first
00061 3 hex digits in a 4 digit Unicode character name; the top column
00062 contains a hex character string of the 4th (low-order) hex digit
00063 of the Unicode character.
00064 */
00065 char *hex[18]= {
00066     "0030:0000000018244242424242424180000", /* Hex digit 0 */
00067     "0031:000000000818280808080808083E0000", /* Hex digit 1 */
00068     "0032:000000003C4242020C102040407E0000", /* Hex digit 2 */
00069     "0033:000000003C4242021C020242423C0000", /* Hex digit 3 */
00070     "0034:00000000040C142444447E0404040000", /* Hex digit 4 */
00071     "0035:000000007E4040407C020202423C0000", /* Hex digit 5 */
00072     "0036:000000001C2040407C424242423C0000", /* Hex digit 6 */
00073     "0037:000000007E02020404080808080000", /* Hex digit 7 */
00074     "0038:000000003C4242423C424242423C0000", /* Hex digit 8 */
00075     "0039:000000003C4242423E02020204380000", /* Hex digit 9 */
00076     "0041:0000000018242442427E424242420000", /* Hex digit A */
00077     "0042:000000007C4242427C424242427C0000", /* Hex digit B */
00078     "0043:000000003C42424040404042423C0000", /* Hex digit C */
00079     "0044:0000000078444242424242424780000", /* Hex digit D */
00080     "0045:000000007E4040407C404040407E0000", /* Hex digit E */
00081     "0046:000000007E4040407C404040400000", /* Hex digit F */
00082     "0055:0000000042424242424242423C0000", /* Unicode 'U' */
00083     "002B:0000000000000808087F080808000000", /* Unicode '+' */
00084 };
00085 unsigned char hexbits[18][32]; ///< The digits converted into bitmaps.
00086
00087 unsigned unipage=0; ///< Unicode page number, 0x00..0xff.
00088 int flip=1; ///< Transpose entire matrix as in Unicode book.
00089
00090
00091 /**
00092  @brief The main function.
00093
00094  @param[in] argc The count of command line arguments.
00095  @param[in] argv Pointer to array of command line arguments.
00096  @return This program exits with status 0.
00097 */
00098 int
00099 main (int argc, char *argv[])
00100 {
00101
00102     int i, j; /* loop variables */
00103     unsigned k0; /* temp Unicode char variable */
00104     unsigned swap; /* temp variable for swapping values */
00105     char inbuf[256]; /* input buffer */
00106     unsigned filesize; /* size of file in bytes */
00107     unsigned bitmapsizes; /* size of bitmap image in bytes */
00108     unsigned thischar; /* the current character */
00109     unsigned char thischarbyte; /* unsigned char lowest byte of Unicode char */
00110     int thischarrow; /* row 0..15 where this character belongs */
00111     int thiscol; /* column 0..15 where this character belongs */
00112     int toppixelrow; /* pixel row, 0..16*32-1 */
00113     unsigned lastpage=0; /* the last Unicode page read in font file */
00114     int wbmp=0; /* set to 1 if writing .wbmp format file */
00115
00116     unsigned char bitmap[17*32][18*4]; /* final bitmap */
00117     unsigned char charbits[32][4]; /* bitmap for one character, 4 bytes/row */
00118
00119     char *infile="", *outfile=""; /* names of input and output files */
00120     FILE *infp, *outfp; /* file pointers of input and output files */
00121
00122     /* initializes bitmap row/col labeling, &c. */
00123     int init (unsigned char bitmap[17*32][18*4]);
00124
00125     /* convert hex string --> bitmap */
00126     int hex2bit (char *instring, unsigned char character[32][4]);
00127
00128     bitmapsizes = 17*32*18*4; /* 17 rows by 18 cols, each 4 bytes */
00129
00130     if (argc > 1) {
00131         for (i = 1; i < argc; i++) {
00132             if (argv[i][0] == '-') { /* this is an option argument */
00133                 switch (argv[i][1]) {
00134                     case 'f': /* flip (transpose) glyphs in bitmap as in standard */
00135                         flip = !flip;
00136                         break;
00137                     case 'i': /* name of input file */
00138                         infile = &argv[i][2];
00139                         break;

```

```

00140         case 'o': /* name of output file */
00141             outfile = &argv[i][2];
00142             break;
00143         case 'p': /* specify a Unicode page other than default of 0 */
00144             sscanf (&argv[i][2], "%x", &unipage); /* Get Unicode page */
00145             break;
00146         case 'w': /* write a .wbmp file instead of a .bmp file */
00147             wbmp = 1;
00148             break;
00149         default: /* if unrecognized option, print list and exit */
00150             fprintf (stderr, "\nSyntax:\n\n");
00151             fprintf (stderr, "  %s -p<Unicode_Page> ", argv[0]);
00152             fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
00153             fprintf (stderr, "  -w specifies .wbmp output instead of ");
00154             fprintf (stderr, "default Windows .bmp output.\n\n");
00155             fprintf (stderr, "  -p is followed by 1 to 6 ");
00156             fprintf (stderr, "Unicode page hex digits ");
00157             fprintf (stderr, "(default is Page 0).\n\n");
00158             fprintf (stderr, "\nExample:\n\n");
00159             fprintf (stderr, "  %s -p83 -iunifont.hex -ou83.bmp\n\n\n",
00160                     argv[0]);
00161             exit (1);
00162     }
00163 }
00164 }
00165 }
00166 /*
00167  Make sure we can open any I/O files that were specified before
00168  doing anything else.
00169 */
00170 if (strlen (infile) > 0) {
00171     if ((infp = fopen (infile, "r")) == NULL) {
00172         fprintf (stderr, "Error: can't open %s for input.\n", infile);
00173         exit (1);
00174     }
00175 }
00176 else {
00177     infp = stdin;
00178 }
00179 if (strlen (outfile) > 0) {
00180     if ((outfp = fopen (outfile, "w")) == NULL) {
00181         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00182         exit (1);
00183     }
00184 }
00185 else {
00186     outfp = stdout;
00187 }
00188
00189 (void)init(bitmap); /* initialize bitmap with row/column headers, etc. */
00190
00191 /*
00192  Read in the characters in the page
00193 */
00194 while (lastpage <= unipage && fgets (inbuf, MAXBUF-1, infp) != NULL) {
00195     sscanf (inbuf, "%x", &thischar);
00196     lastpage = thischar » 8; /* keep Unicode page to see if we can stop */
00197     if (lastpage == unipage) {
00198         thischarbyte = (unsigned char)(thischar & 0xff);
00199         for (k0=0; inbuf[k0] != ':'; k0++);
00200         k0++;
00201         hex2bit (&inbuf[k0], charbits); /* convert hex string to 32*4 bitmap */
00202
00203         /*
00204          Now write character bitmap upside-down in page array, to match
00205          .bmp file order. In the .wbmp' and .bmp files, white is a '1'
00206          bit and black is a '0' bit, so complement charbits[].
00207         */
00208
00209         thiscol = (thischarbyte & 0xf) + 2; /* column number will be 1..16 */
00210         thischarrow = thischarbyte » 4; /* character row number, 0..15 */
00211         if (flip) { /* swap row and column placement */
00212             swap = thiscol;
00213             thiscol = thischarrow;
00214             thischarrow = swap;
00215             thiscol += 2; /* column index starts at 1 */
00216             thischarrow -= 2; /* row index starts at 0 */
00217         }
00218         toppixelrow = 32 * (thischarrow + 1) - 1; /* from bottom to top */
00219
00220         /*

```

```

00221      Copy the center of charbits[] because hex characters only
00222      occupy rows 8 to 23 and column byte 2 (and for 16 bit wide
00223      characters, byte 3). The charbits[] array was given 32 rows
00224      and 4 column bytes for completeness in the beginning.
00225      */
00226      for (i=8; i<24; i++) {
00227          bitmap[toppixelrow + i][(thiscol « 2) | 0] =
00228              ~charbits[i][0] & 0xff;
00229          bitmap[toppixelrow + i][(thiscol « 2) | 1] =
00230              ~charbits[i][1] & 0xff;
00231          bitmap[toppixelrow + i][(thiscol « 2) | 2] =
00232              ~charbits[i][2] & 0xff;
00233          /* Only use first 31 bits; leave vertical rule in 32nd column */
00234          bitmap[toppixelrow + i][(thiscol « 2) | 3] =
00235              ~charbits[i][3] & 0xfe;
00236      }
00237      /*
00238      Leave white space in 32nd column of rows 8, 14, 15, and 23
00239      to leave 16 pixel height upper, middle, and lower guides.
00240      */
00241      bitmap[toppixelrow + 8][(thiscol « 2) | 3] |= 1;
00242      bitmap[toppixelrow + 14][(thiscol « 2) | 3] |= 1;
00243      bitmap[toppixelrow + 15][(thiscol « 2) | 3] |= 1;
00244      bitmap[toppixelrow + 23][(thiscol « 2) | 3] |= 1;
00245  }
00246  }
00247  /*
00248  Now write the appropriate bitmap file format, either
00249  Wireless Bitmap or Microsoft Windows bitmap.
00250  */
00251  if (wbmp) { /* Write a Wireless Bitmap .wbmp format file */
00252      /*
00253      Write WBMP header
00254      */
00255      fprintf (outfp, "%c", 0x00); /* Type of image; always 0 (monochrome) */
00256      fprintf (outfp, "%c", 0x00); /* Reserved; always 0 */
00257      fprintf (outfp, "%c%c", 0x84, 0x40); /* Width = 576 pixels */
00258      fprintf (outfp, "%c%c", 0x84, 0x20); /* Height = 544 pixels */
00259      /*
00260      Write bitmap image
00261      */
00262      for (toppixelrow=0; toppixelrow <= 17*32-1; toppixelrow++) {
00263          for (j=0; j<18; j++) {
00264              fprintf (outfp, "%c", bitmap[toppixelrow][(j«2)  ]);
00265              fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 1]);
00266              fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 2]);
00267              fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 3]);
00268          }
00269      }
00270  }
00271  else { /* otherwise, write a Microsoft Windows .bmp format file */
00272      /*
00273      Write the .bmp file -- start with the header, then write the bitmap
00274      */
00275      /* 'B', 'M' appears at start of every .bmp file */
00276      fprintf (outfp, "%c%c", 0x42, 0x4d);
00277
00278      /* Write file size in bytes */
00279      filesize = 0x3E + bitmapsize;
00280      fprintf (outfp, "%c", (unsigned char)((filesize      ) & 0xff));
00281      fprintf (outfp, "%c", (unsigned char)((filesize » 0x08) & 0xff));
00282      fprintf (outfp, "%c", (unsigned char)((filesize » 0x10) & 0xff));
00283      fprintf (outfp, "%c", (unsigned char)((filesize » 0x18) & 0xff));
00284
00285      /* Reserved - 0's */
00286      fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00287
00288      /* Offset from start of file to bitmap data */
00289      fprintf (outfp, "%c%c%c%c", 0x3E, 0x00, 0x00, 0x00);
00290
00291      /* Length of bitmap info header */
00292      fprintf (outfp, "%c%c%c%c", 0x28, 0x00, 0x00, 0x00);
00293
00294      /* Width of bitmap in pixels */
00295      fprintf (outfp, "%c%c%c%c", 0x40, 0x02, 0x00, 0x00);
00296
00297      /* Height of bitmap in pixels */
00298      fprintf (outfp, "%c%c%c%c", 0x20, 0x02, 0x00, 0x00);
00299
00300      /* Planes in bitmap (fixed at 1) */
00301

```

```

00302     fprintf (outfp, "%c%c", 0x01, 0x00);
00303
00304     /* bits per pixel (1 = monochrome) */
00305     fprintf (outfp, "%c%c", 0x01, 0x00);
00306
00307     /* Compression (0 = none) */
00308     fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00309
00310     /* Size of bitmap data in bytes */
00311     fprintf (outfp, "%c", (unsigned char)((bitmapsizesize >> 0) & 0xff));
00312     fprintf (outfp, "%c", (unsigned char)((bitmapsizesize >> 8) & 0xff));
00313     fprintf (outfp, "%c", (unsigned char)((bitmapsizesize >> 16) & 0xff));
00314     fprintf (outfp, "%c", (unsigned char)((bitmapsizesize >> 24) & 0xff));
00315
00316     /* Horizontal resolution in pixels per meter */
00317     fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);
00318
00319     /* Vertical resolution in pixels per meter */
00320     fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);
00321
00322     /* Number of colors used */
00323     fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);
00324
00325     /* Number of important colors */
00326     fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);
00327
00328     /* The color black: B=0x00, G=0x00, R=0x00, Filler=0xFF */
00329     fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00330
00331     /* The color white: B=0xFF, G=0xFF, R=0xFF, Filler=0xFF */
00332     fprintf (outfp, "%c%c%c%c", 0xFF, 0xFF, 0xFF, 0x00);
00333
00334     /*
00335      * Now write the raw data bits.  Data is written from the lower
00336      * left-hand corner of the image to the upper right-hand corner
00337      * of the image.
00338      */
00339     for (toppixelrow=17*32-1; toppixelrow >= 0; toppixelrow--) {
00340         for (j=0; j<18; j++) {
00341             fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) & 3]);
00342             fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) & 1]);
00343             fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) & 2]);
00344
00345             fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) & 3]);
00346         }
00347     }
00348     exit (0);
00349 }
00350 }
00351
00352 /**
00353  * @brief Generate a bitmap for one glyph.
00354  *
00355  * Convert the portion of a hex string after the ':' into a character bitmap.
00356  *
00357  * If string is >= 128 characters, it will fill all 4 bytes per row.
00358  * If string is >= 64 characters and < 128, it will fill 2 bytes per row.
00359  * Otherwise, it will fill 1 byte per row.
00360  *
00361  * @param[in] instr The character array containing the glyph bitmap.
00362  * @param[out] character Glyph bitmap, 8, 16, or 32 columns by 16 rows tall.
00363  * @return Always returns 0.
00364  */
00365 int
00366 hex2bit (char *instr, unsigned char character[32][4])
00367 {
00368     int i; /* current row in bitmap character */
00369     int j; /* current character in input string */
00370     int k; /* current byte in bitmap character */
00371     int width; /* number of output bytes to fill - 1: 0, 1, 2, or 3 */
00372
00373     for (i=0; i<32; i++) /* erase previous character */
00374         character[i][0] = character[i][1] = character[i][2] = character[i][3] = 0;
00375     j=0; /* current location is at beginning of instr */
00376
00377     if (strlen (instr) <= 34) /* 32 + possible '\r', '\n' */
00378         width = 0;
00379     else if (strlen (instr) <= 66) /* 64 + possible '\r', '\n' */
00380         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 }
00409
00410 /**
00411  @brief Initialize the bitmap grid.
00412  @param[out] bitmap The bitmap to generate, with 32x32 pixel glyph areas.
00413  @return Always returns 0.
00414 */
00415 int
00416 init (unsigned char bitmap[17*32][18*4])
00417 {
00418     int i, j;
00419     unsigned char charbits[32][4]; /* bitmap for one character, 4 bytes/row */
00420     unsigned toppixelrow;
00421     unsigned thiscol;
00422     unsigned char pnybble0, pnybble1, pnybble2, pnybble3;
00423
00424     for (i=0; i<18; i++) { /* bitmaps for '0'..'9', 'A'..'F', 'u', '+' */
00425         hex2bit (&hex[i][5], charbits); /* convert hex string to 32*4 bitmap */
00426
00427         for (j=0; j<32; j++) hexbits[i][j] = ~charbits[j][1];
00428     }
00429
00430     /*
00431      Initialize bitmap to all white.
00432     */
00433     for (toppixelrow=0; toppixelrow < 17*32; toppixelrow++) {
00434         for (thiscol=0; thiscol<18; thiscol++) {
00435             bitmap[toppixelrow][(thiscol << 2) ] = 0xff;
00436             bitmap[toppixelrow][(thiscol << 2) | 1] = 0xff;
00437             bitmap[toppixelrow][(thiscol << 2) | 2] = 0xff;
00438             bitmap[toppixelrow][(thiscol << 2) | 3] = 0xff;
00439         }
00440     }
00441
00442     /*
00443      Write the "u+nnnn" table header in the upper left-hand corner,
00444      where nnnn is the upper 16 bits of a 32-bit Unicode assignment.
00445     */
00446     pnybble3 = (unipage >> 20);
00447     pnybble2 = (unipage >> 16) & 0xf;
00448     pnybble1 = (unipage >> 12) & 0xf;
00449     pnybble0 = (unipage >> 8) & 0xf;
00450
00451     for (i=0; i<32; i++) {
00452         bitmap[i][1] = hexbits[16][i]; /* copy 'u' */
00453         bitmap[i][2] = hexbits[17][i]; /* copy '+' */
00454         bitmap[i][3] = hexbits[pnybble3][i];
00455         bitmap[i][4] = hexbits[pnybble2][i];
00456         bitmap[i][5] = hexbits[pnybble1][i];
00457         bitmap[i][6] = hexbits[pnybble0][i];
00458     }
00459
00460     /*
00461      Write low-order 2 bytes of Unicode number assignments, as hex labels
00462     */
00463     pnybble3 = (unipage >> 4) & 0xf; /* Highest-order hex digit */

```

```

00464 pnybble2 = (unipage ) & 0xf; /* Next highest-order hex digit */
00465 /*
00466  * Write the column headers in bitmap[] (row headers if flipped)
00467  */
00468 toppixelrow = 32 * 17 - 1; /* maximum pixel row number */
00469 /*
00470  * Label the column headers. The hexbits[] bytes are split across two
00471  * bitmap[] entries to center a the hex digits in a column of 4 bytes.
00472  * OR highest byte with 0xf0 and lowest byte with 0xf to make outer
00473  * nybbles white (0=black, 1=white).
00474  */
00475 for (i=0; i<16; i++) {
00476     for (j=0; j<32; j++) {
00477         if (flip) { /* transpose matrix */
00478             bitmap[j][((i+2) < 2) | 0] = (hexbits[pnybble3][j] > 4) | 0xf0;
00479             bitmap[j][((i+2) < 2) | 1] = (hexbits[pnybble3][j] < 4) |
00480                 (hexbits[pnybble2][j] > 4);
00481             bitmap[j][((i+2) < 2) | 2] = (hexbits[pnybble2][j] < 4) |
00482                 (hexbits[i][j] > 4);
00483             bitmap[j][((i+2) < 2) | 3] = (hexbits[i][j] < 4) | 0xf0;
00484         }
00485         else {
00486             bitmap[j][((i+2) < 2) | 1] = (hexbits[i][j] > 4) | 0xf0;
00487             bitmap[j][((i+2) < 2) | 2] = (hexbits[i][j] < 4) | 0xf0;
00488         }
00489     }
00490 }
00491 /*
00492  * Now use the single hex digit column graphics to label the row headers.
00493  */
00494 for (i=0; i<16; i++) {
00495     toppixelrow = 32 * (i + 1) - 1; /* from bottom to top */
00496     for (j=0; j<32; j++) {
00497         if (!flip) { /* if not transposing matrix */
00498             bitmap[toppixelrow + j][4] = hexbits[pnybble3][j];
00499             bitmap[toppixelrow + j][5] = hexbits[pnybble2][j];
00500         }
00501         bitmap[toppixelrow + j][6] = hexbits[i][j];
00502     }
00503 }
00504 /*
00505  * Now draw grid lines in bitmap, around characters we just copied.
00506  */
00507 /* draw vertical lines 2 pixels wide */
00508 for (i=1*32; i<17*32; i++) {
00509     if ((i & 0x1f) == 7)
00510         i++;
00511     else if ((i & 0x1f) == 14)
00512         i += 2;
00513     else if ((i & 0x1f) == 22)
00514         i++;
00515     for (j=1; j<18; j++) {
00516         bitmap[i][(j < 2) | 3] &= 0xfe;
00517     }
00518 }
00519 /* draw horizontal lines 1 pixel tall */
00520 for (i=1*32-1; i<18*32-1; i+=32) {
00521     for (j=2; j<18; j++) {
00522         bitmap[i][(j < 2) ] = 0x00;
00523         bitmap[i][(j < 2) | 1] = 0x81;
00524         bitmap[i][(j < 2) | 2] = 0x81;
00525         bitmap[i][(j < 2) | 3] = 0x00;
00526     }
00527 }
00528 /* fill in top left corner pixel of grid */
00529 bitmap[31][7] = 0xfe;
00530
00531 return (0);
00532 }
00533 }

```

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 164 of file [unihexgen.c](#).

```

00165 {
00166
00167     int grid[16]; /* the glyph grid we'll build */
00168
00169     int row;      /* row number in current glyph */
00170     int digitrow; /* row number in current hex digit being rendered */
00171     int rowbits;  /* 1 & 0 bits to draw current glyph row */
00172
00173     int d1, d2, d3, d4; /* four hexadecimal digits of each code point */
00174
00175     d1 = (thiscp » 12) & 0xF;
00176     d2 = (thiscp » 8) & 0xF;
00177     d3 = (thiscp » 4) & 0xF;
00178     d4 = (thiscp    ) & 0xF;
00179
00180     /* top and bottom rows are white */
00181     grid[0] = grid[15] = 0x0000;
00182
00183     /* 14 inner rows are 14-pixel wide black lines, centered */
00184     for (row = 1; row < 15; row++) grid[row] = 0x7FFE;
00185
00186     printf ("%04X:", thiscp);
00187
00188     /*
00189      * Render the first row of 2 hexadecimal digits
00190      */
00191     digitrow = 0; /* start at top of first row of digits to render */
00192     for (row = 2; row < 7; row++) {
00193         rowbits = (hexdigit[d1][digitrow] « 9) |
00194                 (hexdigit[d2][digitrow] « 3);
00195         grid[row] ^= rowbits; /* digits appear as white on black background */
00196         digitrow++;
00197     }
00198
00199     /*
00200      * Render the second row of 2 hexadecimal digits
00201      */
00202     digitrow = 0; /* start at top of first row of digits to render */
00203     for (row = 9; row < 14; row++) {
00204         rowbits = (hexdigit[d3][digitrow] « 9) |
00205                 (hexdigit[d4][digitrow] « 3);
00206         grid[row] ^= rowbits; /* digits appear as white on black background */
00207         digitrow++;
00208     }
00209
00210     for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);
00211
00212     putchar ('\n');
00213
00214     return;
00215 }

```

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 227 of file [unihexgen.c](#).

```

00228 {
00229
00230     int grid[16]; /* the glyph grid we'll build */
00231
00232     int row;      /* row number in current glyph */
00233     int digitrow; /* row number in current hex digit being rendered */
00234     int rowbits;  /* 1 & 0 bits to draw current glyph row */
00235

```

```

00236 int d1, d2, d3, d4, d5, d6; /* six hexadecimal digits of each code point */
00237
00238 d1 = (thiscp » 20) & 0xF;
00239 d2 = (thiscp » 16) & 0xF;
00240 d3 = (thiscp » 12) & 0xF;
00241 d4 = (thiscp » 8) & 0xF;
00242 d5 = (thiscp » 4) & 0xF;
00243 d6 = (thiscp ) & 0xF;
00244
00245 /* top and bottom rows are white */
00246 grid[0] = grid[15] = 0x0000;
00247
00248 /* 14 inner rows are 16-pixel wide black lines, centered */
00249 for (row = 1; row < 15; row++) grid[row] = 0xFFFF;
00250
00251 printf ("%06X:", thiscp);
00252
00253 /*
00254  * Render the first row of 3 hexadecimal digits
00255  */
00256 digitrow = 0; /* start at top of first row of digits to render */
00257 for (row = 2; row < 7; row++) {
00258     rowbits = (hexdigit[d1][digitrow] « 11) |
00259             (hexdigit[d2][digitrow] « 6) |
00260             (hexdigit[d3][digitrow] « 1);
00261     grid[row] ^= rowbits; /* digits appear as white on black background */
00262     digitrow++;
00263 }
00264
00265 /*
00266  * Render the second row of 3 hexadecimal digits
00267  */
00268 digitrow = 0; /* start at top of first row of digits to render */
00269 for (row = 9; row < 14; row++) {
00270     rowbits = (hexdigit[d4][digitrow] « 11) |
00271             (hexdigit[d5][digitrow] « 6) |
00272             (hexdigit[d6][digitrow] « 1);
00273     grid[row] ^= rowbits; /* digits appear as white on black background */
00274     digitrow++;
00275 }
00276
00277 for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);
00278
00279 putchar ('\n');
00280
00281 return;
00282 }
00283

```

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 116 of file [unihexgen.c](#).

```

00117 {
00118
00119 unsigned startcp, endcp, thiscp;
00120 void hexprint4(int); /* function to print one 4-digit unifont.hex code point */
00121 void hexprint6(int); /* function to print one 6-digit unifont.hex code point */

```

```

00122
00123 if (argc != 3) {
00124     fprintf(stderr, "\n%s - generate unifont.hex code points as\n", argv[0]);
00125     fprintf(stderr, "four-digit hexadecimal numbers in a 2 by 2 grid.\n");
00126     fprintf(stderr, "or six-digit hexadecimal numbers in a 3 by 2 grid.\n");
00127     fprintf(stderr, "Syntax:\n\n");
00128     fprintf(stderr, "    %s first_code_point last_code_point > glyphs.hex\n\n", argv[0]);
00129     fprintf(stderr, "Example (to generate glyphs for the Private Use Area):\n\n");
00130     fprintf(stderr, "    %s e000 f8ff > pua.hex\n\n", argv[0]);
00131     exit (EXIT_FAILURE);
00132 }
00133
00134 sscanf (argv[1], "%x", &startcp);
00135 sscanf (argv[2], "%x", &endcp);
00136
00137 startcp &= 0xFFFFF; /* limit to 6 hex digits */
00138 endcp   &= 0xFFFFF; /* limit to 6 hex digits */
00139
00140 /*
00141  * For each code point in the desired range, generate a glyph.
00142  */
00143 for (thiscp = startcp; thiscp <= endcp; thiscp++) {
00144     if (thiscp <= 0xFFFF) {
00145         hexprint4 (thiscp); /* print digits 2/line, 2 lines */
00146     }
00147     else {
00148         hexprint6 (thiscp); /* print digits 3/line, 2 lines */
00149     }
00150 }
00151 exit (EXIT_SUCCESS);
00152 }

```

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 88 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.
00026
00027  This utility program works in Roman Czyborra's unifont.hex file
00028  format, the basis of the GNU Unifont package.
00029 */
00030 /*
00031  This program is released under the terms of the GNU General Public
00032  License version 2, or (at your option) a later version.
00033
00034  LICENSE:
00035
00036  This program is free software: you can redistribute it and/or modify
00037  it under the terms of the GNU General Public License as published by
00038  the Free Software Foundation, either version 2 of the License, or
00039  (at your option) any later version.
00040
00041  This program is distributed in the hope that it will be useful,
00042  but WITHOUT ANY WARRANTY; without even the implied warranty of
00043  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00044  GNU General Public License for more details.
00045
00046  You should have received a copy of the GNU General Public License
00047  along with this program. If not, see <http://www.gnu.org/licenses/>.
00048 */
00049 /*
00050  6 September 2025 [Paul Hardy]:
00051  - Changed startcp, endcp, and thiscp from "int" to "unsigned"
00052    for compatibility with sscanf definition.
00053 */
00054 #include <stdio.h>
00055 #include <stdlib.h>
00056
00057 /**
00058  @brief Bitmap pattern for each hexadecimal digit.
00059
00060  hexdigit[][] definition: the bitmap pattern for
00061  each hexadecimal digit.
00062
00063  Each digit is drawn as a 4 wide by 5 high bitmap,
00064  so each digit row is one hexadecimal digit, and
00065  each entry has 5 rows.
00066
00067  For example, the entry for digit 1 is:
00068
00069      {0x2,0x6,0x2,0x2,0x7},
00070
```

```

00071
00072 which corresponds graphically to:
00073
00074 --#- ==> 0010 ==> 0x2
00075 -##- ==> 0110 ==> 0x6
00076 --#- ==> 0010 ==> 0x2
00077 --#- ==> 0010 ==> 0x2
00078 -### ==> 0111 ==> 0x7
00079
00080 These row values will then be exclusive-ORed with four one bits
00081 (binary 1111, or 0xF) to form white digits on a black background.
00082
00083
00084 Functions hexprint4 and hexprint6 share the hexdigit array;
00085 they print four-digit and six-digit hexadecimal code points
00086 in a single glyph, respectively.
00087 */
00088 char hexdigit[16][5] = {
00089     {0x6,0x9,0x9,0x9,0x6}, /* 0x0 */
00090     {0x2,0x6,0x2,0x2,0x7}, /* 0x1 */
00091     {0xF,0x1,0xF,0x8,0xF}, /* 0x2 */
00092     {0xE,0x1,0x7,0x1,0xE}, /* 0x3 */
00093     {0x9,0x9,0xF,0x1,0x1}, /* 0x4 */
00094     {0xF,0x8,0xF,0x1,0xF}, /* 0x5 */
00095     {0x6,0x8,0xE,0x9,0x6}, /* 0x6 */ // {0x8,0x8,0xF,0x9,0xF} [alternate square form of 6]
00096     {0xF,0x1,0x2,0x4,0x4}, /* 0x7 */
00097     {0x6,0x9,0x6,0x9,0x6}, /* 0x8 */
00098     {0x6,0x9,0x7,0x1,0x6}, /* 0x9 */ // {0xF,0x9,0xF,0x1,0x1} [alternate square form of 9]
00099     {0xF,0x9,0xF,0x9,0x9}, /* 0xA */
00100     {0xE,0x9,0xE,0x9,0xE}, /* 0xB */
00101     {0x7,0x8,0x8,0x8,0x7}, /* 0xC */
00102     {0xE,0x9,0x9,0x9,0xE}, /* 0xD */
00103     {0xF,0x8,0xE,0x8,0xF}, /* 0xE */
00104     {0xF,0x8,0xE,0x8,0x8} /* 0xF */
00105 };
00106
00107
00108 /**
00109  @brief The main function.
00110
00111  @param[in] argc The count of command line arguments.
00112  @param[in] argv Pointer to array of command line arguments (code point range).
00113  @return This program exits with status EXIT_SUCCESS.
00114  */
00115 int
00116 main (int argc, char *argv[])
00117 {
00118
00119     unsigned startcp, endcp, thiscp;
00120     void hexprint4(int); /* function to print one 4-digit unifont.hex code point */
00121     void hexprint6(int); /* function to print one 6-digit unifont.hex code point */
00122
00123     if (argc != 3) {
00124         fprintf (stderr, "\n%s - generate unifont.hex code points as\n", argv[0]);
00125         fprintf (stderr, "four-digit hexadecimal numbers in a 2 by 2 grid.\n");
00126         fprintf (stderr, "or six-digit hexadecimal numbers in a 3 by 2 grid.\n");
00127         fprintf (stderr, "Syntax:\n\n");
00128         fprintf (stderr, "    %s first_code_point last_code_point > glyphs.hex\n\n", argv[0]);
00129         fprintf (stderr, "Example (to generate glyphs for the Private Use Area):\n\n");
00130         fprintf (stderr, "    %s e000 f8ff > pua.hex\n\n", argv[0]);
00131         exit (EXIT_FAILURE);
00132     }
00133
00134     sscanf (argv[1], "%x", &startcp);
00135     sscanf (argv[2], "%x", &endcp);
00136
00137     startcp &= 0xFFFFF; /* limit to 6 hex digits */
00138     endcp &= 0xFFFFF; /* limit to 6 hex digits */
00139
00140     /*
00141      For each code point in the desired range, generate a glyph.
00142     */
00143     for (thiscp = startcp; thiscp <= endcp; thiscp++) {
00144         if (thiscp <= 0xFFFF) {
00145             hexprint4 (thiscp); /* print digits 2/line, 2 lines */
00146         }
00147         else {
00148             hexprint6 (thiscp); /* print digits 3/line, 2 lines */
00149         }
00150     }
00151     exit (EXIT_SUCCESS);

```



```

00152 }
00153
00154
00155 /**
00156  @brief Generate a bitmap containing a 4-digit Unicode code point.
00157
00158  Takes a 4-digit Unicode code point as an argument
00159  and prints a unifont.hex string for it to stdout.
00160
00161  @param[in] thiscp The current code point for which to generate a glyph.
00162 */
00163 void
00164 hexprint4 (int thiscp)
00165 {
00166
00167     int grid[16]; /* the glyph grid we'll build */
00168
00169     int row;      /* row number in current glyph */
00170     int digitrow; /* row number in current hex digit being rendered */
00171     int rowbits;  /* 1 & 0 bits to draw current glyph row */
00172
00173     int d1, d2, d3, d4; /* four hexadecimal digits of each code point */
00174
00175     d1 = (thiscp » 12) & 0xF;
00176     d2 = (thiscp » 8) & 0xF;
00177     d3 = (thiscp » 4) & 0xF;
00178     d4 = (thiscp ) & 0xF;
00179
00180     /* top and bottom rows are white */
00181     grid[0] = grid[15] = 0x0000;
00182
00183     /* 14 inner rows are 14-pixel wide black lines, centered */
00184     for (row = 1; row < 15; row++) grid[row] = 0x7FFE;
00185
00186     printf ("%04X:", thiscp);
00187
00188     /*
00189      Render the first row of 2 hexadecimal digits
00190      */
00191     digitrow = 0; /* start at top of first row of digits to render */
00192     for (row = 2; row < 7; row++) {
00193         rowbits = (hexdigit[d1][digitrow] « 9) |
00194                 (hexdigit[d2][digitrow] « 3);
00195         grid[row] ^= rowbits; /* digits appear as white on black background */
00196         digitrow++;
00197     }
00198
00199     /*
00200      Render the second row of 2 hexadecimal digits
00201      */
00202     digitrow = 0; /* start at top of first row of digits to render */
00203     for (row = 9; row < 14; row++) {
00204         rowbits = (hexdigit[d3][digitrow] « 9) |
00205                 (hexdigit[d4][digitrow] « 3);
00206         grid[row] ^= rowbits; /* digits appear as white on black background */
00207         digitrow++;
00208     }
00209
00210     for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);
00211
00212     putchar ('\n');
00213
00214     return;
00215 }
00216
00217
00218 /**
00219  @brief Generate a bitmap containing a 6-digit Unicode code point.
00220
00221  Takes a 6-digit Unicode code point as an argument
00222  and prints a unifont.hex string for it to stdout.
00223
00224  @param[in] thiscp The current code point for which to generate a glyph.
00225 */
00226 void
00227 hexprint6 (int thiscp)
00228 {
00229
00230     int grid[16]; /* the glyph grid we'll build */
00231
00232     int row;      /* row number in current glyph */

```

```

00233 int digitrow; /* row number in current hex digit being rendered */
00234 int rowbits; /* 1 & 0 bits to draw current glyph row */
00235
00236 int d1, d2, d3, d4, d5, d6; /* six hexadecimal digits of each code point */
00237
00238 d1 = (thiscp » 20) & 0xF;
00239 d2 = (thiscp » 16) & 0xF;
00240 d3 = (thiscp » 12) & 0xF;
00241 d4 = (thiscp » 8) & 0xF;
00242 d5 = (thiscp » 4) & 0xF;
00243 d6 = (thiscp ) & 0xF;
00244
00245 /* top and bottom rows are white */
00246 grid[0] = grid[15] = 0x0000;
00247
00248 /* 14 inner rows are 16-pixel wide black lines, centered */
00249 for (row = 1; row < 15; row++) grid[row] = 0xFFFF;
00250
00251 printf ("%06X:", thiscp);
00252
00253 /*
00254  * Render the first row of 3 hexadecimal digits
00255  */
00256 digitrow = 0; /* start at top of first row of digits to render */
00257 for (row = 2; row < 7; row++) {
00258     rowbits = (hexdigit[d1][digitrow] « 11) |
00259             (hexdigit[d2][digitrow] « 6) |
00260             (hexdigit[d3][digitrow] « 1);
00261     grid[row] ^= rowbits; /* digits appear as white on black background */
00262     digitrow++;
00263 }
00264
00265 /*
00266  * Render the second row of 3 hexadecimal digits
00267  */
00268 digitrow = 0; /* start at top of first row of digits to render */
00269 for (row = 9; row < 14; row++) {
00270     rowbits = (hexdigit[d4][digitrow] « 11) |
00271             (hexdigit[d5][digitrow] « 6) |
00272             (hexdigit[d6][digitrow] « 1);
00273     grid[row] ^= rowbits; /* digits appear as white on black background */
00274     digitrow++;
00275 }
00276
00277 for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);
00278
00279 putchar ('\n');
00280
00281 return;
00282 }
00283
00284

```

5.39 unihexpose.c

```

00001 /**
00002  * @file: unihetranspose.c
00003
00004  * @brief: Transpose Unifont glyph bitmaps.
00005
00006  * This program takes Unifont .hex format glyphs and converts those
00007  * glyphs so that each byte (two hexadecimal digits in the .hex file)
00008  * represents a column of 8 rows. This simplifies use with graphics
00009  * display controllers that write lines consisting of 8 rows at a time
00010  * to a display.
00011
00012  * The bytes are ordered as first all the columns for the glyph in
00013  * the first 8 rows, then all the columns in the next 8 rows, with
00014  * columns ordered from left to right.
00015
00016  * This file must be linked with functions in unifont-support.c.
00017
00018  * @author Paul Hardy
00019
00020  * @copyright Copyright © 2023 Paul Hardy
00021  */
00022 /*
00023  * LICENSE:

```

```

00024
00025     This program is free software: you can redistribute it and/or modify
00026     it under the terms of the GNU General Public License as published by
00027     the Free Software Foundation, either version 2 of the License, or
00028     (at your option) any later version.
00029
00030     This program is distributed in the hope that it will be useful,
00031     but WITHOUT ANY WARRANTY; without even the implied warranty of
00032     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00033     GNU General Public License for more details.
00034
00035     You should have received a copy of the GNU General Public License
00036     along with this program. If not, see <http://www.gnu.org/licenses/>.
00037 */
00038 #include <stdio.h>
00039 #include <stdlib.h>
00040
00041 #define MAXWIDTH 128
00042
00043 int
00044 main (int argc, char *argv[]) {
00045     unsigned codept; /* Unicode code point for glyph */
00046     char instring [MAXWIDTH]; /* input Unifont hex string */
00047     char outstring [MAXWIDTH]; /* output Unifont hex string */
00048     int width; /* width of current glyph */
00049     unsigned char glyph [16][2];
00050     unsigned char glyphbits [16][16]; /* One glyphbits row, for transposing */
00051     unsigned char transpose [2][16]; /* Transposed glyphbits bitmap */
00052
00053     void print_syntax (void);
00054
00055     void parse_hex (char *hexstring,
00056                    int *width,
00057                    unsigned *codept,
00058                    unsigned char glyph[16][2]);
00059
00060     void glyph2bits (int width,
00061                     unsigned char glyph[16][2],
00062                     unsigned char glyphbits [16][16]);
00063
00064     void hexpose (int width,
00065                  unsigned char glyphbits [16][16],
00066                  unsigned char transpose [2][16]);
00067
00068     void xglyph2string (int width, unsigned codept,
00069                        unsigned char transpose [2][16],
00070                        char *outstring);
00071
00072     if (argc > 1) {
00073         print_syntax ();
00074         exit (EXIT_FAILURE);
00075     }
00076
00077     while (fgets (instring, MAXWIDTH, stdin) != NULL) {
00078         parse_hex (instring, &width, &codept, glyph);
00079
00080         glyph2bits (width, glyph, glyphbits);
00081
00082         hexpose (width, glyphbits, transpose);
00083
00084         xglyph2string (width, codept, transpose, outstring);
00085
00086         fprintf (stdout, "%s\n", outstring);
00087     }
00088
00089     exit (EXIT_SUCCESS);
00090 }
00091
00092 void
00093 print_syntax (void) {
00094     fprintf (stderr, "\nSyntax: unihexpose < input.hex > output.hex\n\n");
00095
00096     return;
00097 }
00098
00099 }
00100

```

5.40 src/unijohab2html.c File Reference

Display overlapped 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 0xFFFFFFFF`
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 overlapped 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
00099      variations, there will be no overlap. Then go through
00100      choseong, and then jongseong to find the overlapping
00101      combinations. This saves storage space as an alternative
00102      to storing large 2- or 3-dimensional overlap matrices.
00103     */
00104     // jungcho: Jungseong overlap with Choseong
00105     unsigned jungcho [TOTAL_JUNG * JUNG_VARIATIONS];
00106     // jongjung: Jongseong overlap with Jungseong -- for future expansion
00107     // unsigned jongjung [TOTAL_JUNG * JUNG_VARIATIONS];
00108
00109     int glyphs_overlap; /* If glyph pair being considered overlap. */
00110     int cho_overlaps = 0; /* Number of choseong+vowel overlaps. */
00111     // int jongjung_overlaps = 0; /* Number of vowel+jongseong overlaps. */
00112
00113     int inindex = 0;
00114     int outindex = 0;
00115     FILE *infp, *outfp; /* Input and output file pointers. */
00116
00117     void    parse_args (int argc, char *argv[], int *inindex, int *outindex,
00118                        int *modern_only);
00119     int    cho_variation (int cho, int jung, int jong);
00120     unsigned hangul_read_base16 (FILE *infp, unsigned glyph[][16]);
00121     int    glyph_overlap (unsigned *glyph1, unsigned *glyph2);
00122
00123     void    combine_glyphs (unsigned *glyph1, unsigned *glyph2,
00124                           unsigned *combined_glyph);
00125     void    print_glyph_txt (FILE *fp, unsigned codept, unsigned *this_glyph);
00126
00127     /*
00128      Parse command line arguments to open input & output files, if given.
00129     */
00130     if (argc > 1) {
00131         parse_args (argc, argv, &inindex, &outindex, &modern_only);
00132     }
00133
00134     if (inindex == 0) {
00135         infp = stdin;
00136     }
00137     else {
00138         infp = fopen (argv[inindex], "r");
00139         if (infp == NULL) {
00140             fprintf (stderr, "\n*** ERROR: Cannot open %s for input.\n\n",
00141
```

```

00142         argv[inindex]);
00143     exit (EXIT_FAILURE);
00144 }
00145 }
00146 if (outindex == 0) {
00147     outfp = stdout;
00148 }
00149 else {
00150     outfp = fopen (argv[outindex], "w");
00151     if (outfp == NULL) {
00152         fprintf (stderr, "\n*** ERROR: Cannot open %s for output.\n\n",
00153             argv[outindex]);
00154         exit (EXIT_FAILURE);
00155     }
00156 }
00157
00158 /*
00159  Initialize glyph array to all zeroes.
00160 */
00161 for (codept = 0; codept < MAX_GLYPHS; codept++) {
00162     for (i = 0; i < 16; i++) glyph[codept][i] = 0x0000;
00163 }
00164
00165 /*
00166  Initialize overlap matrices to all zeroes.
00167 */
00168 for (i = 0; i < TOTAL_JUNG * JUNG_VARIATIONS; i++) {
00169     jungcho [i] = 0;
00170 }
00171 // jongjung is reserved for expansion.
00172 // for (i = 0; i < TOTAL_JONG * JONG_VARIATIONS; i++) {
00173 //     jongjung [i] = 0;
00174 // }
00175
00176 /*
00177  Read Hangul base glyph file.
00178 */
00179 max_codept = hangul_read_base16 (infp, glyph);
00180 if (max_codept > 0x8FF) {
00181     fprintf (stderr, "\nWARNING: Hangul glyph range exceeds PUA space.\n\n");
00182 }
00183
00184 /*
00185  If only examining modern Hangul, fill the ancient glyphs
00186  with blanks to guarantee they won't overlap. This is
00187  not as efficient as ending loops sooner, but is easier
00188  to verify for correctness.
00189 */
00190 if (modern_only) {
00191     for (i = 0x0073; i < JUNG_HEX; i++) {
00192         for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
00193     }
00194     for (i = 0x027A; i < JONG_HEX; i++) {
00195         for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
00196     }
00197     for (i = 0x032B; i < 0x0400; i++) {
00198         for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
00199     }
00200 }
00201
00202 /*
00203  Initialize grids to all black (no color) for each of
00204  the 12 Choseong groups.
00205 */
00206 for (group = 0; group < 12; group++) {
00207     for (i = 0; i < 16; i++) {
00208         for (j = 0; j < 16; j++) {
00209             grid[group][i][j] = BLACK; /* No color at first */
00210         }
00211     }
00212 }
00213
00214 /*
00215  Superimpose all Choseong glyphs according to group.
00216  Each grid spot with choseong will be blue.
00217 */
00218 for (group = 0; group < 6; group++) {
00219     for (consonant1 = CHO_HEX + group;
00220         consonant1 < CHO_HEX +
00221             CHO_VARIATIONS * TOTAL_CHO;
00222         consonant1 += CHO_VARIATIONS) {

```

```

00223     for (i = 0; i < 16; i++) { /* For each glyph row */
00224         mask = 0x8000;
00225         for (j = 0; j < 16; j++) {
00226             if (glyph[consonant1][i] & mask) grid[group][i][j] |= BLUE;
00227             mask >>= 1; /* Get next bit in glyph row */
00228         }
00229     }
00230 }
00231 }
00232
00233 /*
00234  Fill with Choseong (initial consonant) to prepare
00235  for groups 3-5 with jongseong except niuen (group+3),
00236  then for groups 3-5 with jongseong nieun (group+6).
00237 */
00238 for (group = 3; group < 6; group++) {
00239     for (i = 0; i < 16; i++) {
00240         for (j = 0; j < 16; j++) {
00241             grid[group + 6][i][j] = grid[group + 3][i][j]
00242                 = grid[group][i][j];
00243         }
00244     }
00245 }
00246
00247 /*
00248  For each Jungseong, superimpose first variation on
00249  appropriate Choseong group for grids 0 to 5.
00250 */
00251 for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
00252     group = cho_variation (-1, vowel, -1);
00253     glyphs_overlap = 0; /* Assume the 2 glyphs do not overlap. */
00254
00255     for (i = 0; i < 16; i++) { /* For each glyph row */
00256         mask = 0x8000;
00257         for (j = 0; j < 16; j++) {
00258             if (glyph[JUNG_HEX + JUNG_VARIATIONS * vowel][i] & mask) {
00259                 /*
00260                  If there was already blue in this grid cell,
00261                  mark this vowel variation as having overlap
00262                  with choseong (initial consonant) letter(s).
00263                 */
00264                 if (grid[group][i][j] & BLUE) glyphs_overlap = 1;
00265
00266                 /* Add green to grid cell color. */
00267                 grid[group][i][j] |= GREEN;
00268             }
00269             mask >>= 1; /* Mask for next bit in glyph row */
00270         } /* for j */
00271     } /* for i */
00272     if (glyphs_overlap) {
00273         jungcho [JUNG_VARIATIONS * vowel] = 1;
00274         cho_overlaps++;
00275     }
00276 } /* for each vowel */
00277
00278 /*
00279  For each Jungseong, superimpose second variation on
00280  appropriate Choseong group for grids 6 to 8.
00281 */
00282 for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
00283     /*
00284      The second vowel variation is for combination with
00285      a final consonant (Jongseong), with initial consonant
00286      (Choseong) variations (or "groups") 3 to 5. Thus,
00287      if the vowel type returns an initial Choseong group
00288      of 0 to 2, add 3 to it.
00289     */
00290     group = cho_variation (-1, vowel, -1);
00291     /*
00292      Groups 0 to 2 don't use second vowel variation,
00293      so increment if group is below 2.
00294     */
00295     if (group < 3) group += 3;
00296     glyphs_overlap = 0; /* Assume the 2 glyphs do not overlap. */
00297
00298     for (i = 0; i < 16; i++) { /* For each glyph row */
00299         mask = 0x8000; /* Start mask at leftmost glyph bit */
00300         for (j = 0; j < 16; j++) { /* For each column in this row */
00301             /* "+ 1" is to get each vowel's second variation */
00302             if (glyph [JUNG_HEX +
00303                     JUNG_VARIATIONS * vowel + 1][i] & mask) {

```



```

00304         /* If this cell has blue already, mark as overlapped. */
00305         if (grid [group + 3][i][j] & BLUE) glyphs_overlap = 1;
00306
00307         /* Superimpose green on current cell color. */
00308         grid [group + 3][i][j] |= GREEN;
00309     }
00310     mask »= 1; /* Get next bit in glyph row */
00311 } /* for j */
00312 } /* for i */
00313 if (glyphs_overlap) {
00314     jungcho [JUNG_VARIATIONS * vowel + 1] = 1;
00315     cho_overlaps++;
00316 }
00317 } /* for each vowel */
00318
00319 /*
00320  For each Jungseong, superimpose third variation on
00321  appropriate Choseong group for grids 9 to 11 for
00322  final consonant (Jongseong) of Nieun.
00323 */
00324 for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
00325     group = cho_variation (-1, vowel, -1);
00326     if (group < 3) group += 3;
00327     glyphs_overlap = 0; /* Assume the 2 glyphs do not overlap. */
00328
00329     for (i = 0; i < 16; i++) { /* For each glyph row */
00330         mask = 0x8000;
00331         for (j = 0; j < 16; j++) {
00332             if (glyph[JUNG_HEX +
00333                     JUNG_VARIATIONS * vowel + 2][i] & mask) {
00334                 /* If this cell has blue already, mark as overlapped. */
00335                 if (grid[group + 6][i][j] & BLUE) glyphs_overlap = 1;
00336
00337                 grid[group + 6][i][j] |= GREEN;
00338             }
00339             mask »= 1; /* Get next bit in glyph row */
00340         } /* for j */
00341     } /* for i */
00342     if (glyphs_overlap) {
00343         jungcho [JUNG_VARIATIONS * vowel + 2] = 1;
00344         cho_overlaps++;
00345     }
00346 } /* for each vowel */
00347
00348
00349 /*
00350  Superimpose all final consonants except nieun for grids 6 to 8.
00351 */
00352 for (consonant2 = 0; consonant2 < TOTAL_JONG; consonant2++) {
00353     /*
00354      Skip over Jongseong Nieun, because it is covered in
00355      grids 9 to 11 after this loop.
00356     */
00357     if (consonant2 == 3) consonant2++;
00358
00359     glyphs_overlap = 0; /* Assume the 2 glyphs do not overlap. */
00360     for (i = 0; i < 16; i++) { /* For each glyph row */
00361         mask = 0x8000;
00362         for (j = 0; j < 16; j++) {
00363             if (glyph[JONG_HEX +
00364                     JONG_VARIATIONS * consonant2][i] & mask) {
00365                 if (grid[6][i][j] & GREEN ||
00366                     grid[7][i][j] & GREEN ||
00367                     grid[8][i][j] & GREEN) glyphs_overlap = 1;
00368
00369                 grid[6][i][j] |= RED;
00370                 grid[7][i][j] |= RED;
00371                 grid[8][i][j] |= RED;
00372             }
00373             mask »= 1; /* Get next bit in glyph row */
00374         } /* for j */
00375     } /* for i */
00376     // jongjung is for expansion
00377     // if (glyphs_overlap) {
00378     //     jongjung [JONG_VARIATIONS * consonant2] = 1;
00379     //     jongjung_overlaps++;
00380     // }
00381 } /* for each final consonant except nieun */
00382
00383 /*
00384  Superimpose final consonant 3 (Jongseong Nieun) on

```

```

00385     groups 9 to 11.
00386 */
00387 codept = JONG_HEX + 3 * JONG_VARIATIONS;
00388
00389 for (i = 0; i < 16; i++) { /* For each glyph row */
00390     mask = 0x8000;
00391     for (j = 0; j < 16; j++) {
00392         if (glyph[codept][i] & mask) {
00393             grid[ 9][i][j] |= RED;
00394             grid[10][i][j] |= RED;
00395             grid[11][i][j] |= RED;
00396         }
00397         mask »= 1; /* Get next bit in glyph row */
00398     }
00399 }
00400
00401
00402 /*
00403  Turn the black (uncolored) cells into white for better
00404  visibility of grid when displayed.
00405 */
00406 for (group = 0; group < 12; group++) {
00407     for (i = 0; i < 16; i++) {
00408         for (j = 0; j < 16; j++) {
00409             if (grid[group][i][j] == BLACK) grid[group][i][j] = WHITE;
00410         }
00411     }
00412 }
00413
00414
00415 /*
00416  Generate HTML output.
00417 */
00418 fprintf (outfp, "<html>\n");
00419 fprintf (outfp, "<head>\n");
00420 fprintf (outfp, "    <title>Johab 6/3/1 Overlaps</title>\n");
00421 fprintf (outfp, "</head>\n");
00422 fprintf (outfp, "<body bgcolor=\"#FFFFCC\">\n");
00423
00424 fprintf (outfp, "<center>\n");
00425 fprintf (outfp, "    <h1>Unifont Hangul Jamo Syllable Components</h1>\n");
00426 fprintf (outfp, "    <h2>Johab 6/3/1 Overlap</h2><br><br>\n");
00427
00428 /* Print the color code key for the table. */
00429 fprintf (outfp, "    <table border=\"1\" cellpadding=\"10\">\n");
00430 fprintf (outfp, "        <tr><th colspan=\"2\" align=\"center\" bgcolor=\"#FFCC80\"></th></tr>\n");
00431 fprintf (outfp, "        <font size=\"+1\">Key</font></th></tr>\n");
00432 fprintf (outfp, "        <tr>\n");
00433 fprintf (outfp, "            <th align=\"center\" bgcolor=\"#FFF800\">Color</th>\n");
00434 fprintf (outfp, "            <th align=\"center\" bgcolor=\"#FFF800\">Letter(s)</th>\n");
00435 fprintf (outfp, "        </tr>\n");
00436
00437 fprintf (outfp, "        <tr><td bgcolor=\"#06X\">, BLUE);
00438 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00439 fprintf (outfp, "<td>Choseong (Initial Consonant)</td></tr>\n");
00440
00441 fprintf (outfp, "        <tr><td bgcolor=\"#06X\">, GREEN);
00442 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00443 fprintf (outfp, "<td>Jungseong (Medial Vowel/Diphthong)</td></tr>\n");
00444
00445 fprintf (outfp, "        <tr><td bgcolor=\"#06X\">, RED);
00446 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00447 fprintf (outfp, "<td>Jongseong (Final Consonant)</td></tr>\n");
00448
00449 fprintf (outfp, "        <tr><td bgcolor=\"#06X\">, BLUE | GREEN);
00450 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00451 fprintf (outfp, "<td>Choseong + Jungseong Overlap</td></tr>\n");
00452
00453 fprintf (outfp, "        <tr><td bgcolor=\"#06X\">, GREEN | RED);
00454 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00455 fprintf (outfp, "<td>Jungseong + Jongseong Overlap</td></tr>\n");
00456
00457 fprintf (outfp, "        <tr><td bgcolor=\"#06X\">, RED | BLUE);
00458 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00459 fprintf (outfp, "<td>Choseong + Jongseong Overlap</td></tr>\n");
00460
00461 fprintf (outfp, "        <tr><td bgcolor=\"#06X\">, RED | GREEN | BLUE);
00462 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00463 fprintf (outfp, "<td>Choseong + Jungseong + Jongseong Overlap</td></tr>\n");
00464
00465 fprintf (outfp, "    </table>\n");

```

Generated by Doxygen

```

00547     group.
00548     */
00549     if (vowel_variation > 0 && group < 3) group += 3;
00550
00551     for (consonant1 = 0; consonant1 < TOTAL_CHO; consonant1++) {
00552         overlapped = glyph_overlap (glyph [i],
00553             glyph [consonant1 * CHO_VARIATIONS
00554                 + CHO_HEX + group]);
00555
00556         /*
00557          * If we just entered ancient choseong range, flag it.
00558          */
00559         if (overlapped && consonant1 >= 19 && ancient_choseong == 0) {
00560             fprintf (outfp, "<font color=\"#0000FF\"><b>");
00561             fprintf (outfp, "&hellip;Ancient Choseong&hellip;</b></font>\n");
00562             ancient_choseong = 1;
00563         }
00564         /*
00565          * If overlapping choseong found, print combined glyph.
00566          */
00567         if (overlapped != 0) {
00568             combine_glyphs (glyph [i],
00569                 glyph [consonant1 * CHO_VARIATIONS
00570                     + CHO_HEX + group],
00571                 tmp_glyph);
00572
00573             print_glyph_txt (outfp,
00574                 PUA_START +
00575                 consonant1 * CHO_VARIATIONS +
00576                 CHO_HEX + group,
00577                 tmp_glyph);
00578
00579         } /* If overlapping pixels found. */
00580     } /* For each initial consonant (Choseong) */
00581 } /* Find the initial consonant that overlapped this vowel variation. */
00582 } /* For each variation of each vowel (Jungseong) */
00583
00584 fputc ('\n', outfp);
00585
00586 fprintf (outfp, "</pre></font>\n");
00587 fprintf (outfp, "</body>\n");
00588 fprintf (outfp, "</html>\n");
00589
00590 fclose (infp);
00591 fclose (outfp);
00592
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 strcmp (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 (strcmp (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 (strcmp (argv [arg_count], "-m", 2) == 0 ||
00627             strcmp (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 (strcmp (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 (strcmp (argv [arg_count], "-h", 2) == 0 ||
00639             strcmp (argv [arg_count], "--help", 6) == 0) {
00640             printf ("\nunjohab2html [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");
00647             printf ("    -i      input_file  Unifont hangul-base.hex formatted input file.\n\n");
00648             printf ("    -o      output_file  HTML output file showing overlapping letters.\n\n");
00649             printf ("    -m, --modern      Only examine modern Hangul letters.\n\n");
00650             printf ("    Example:\n\n");
00651             printf ("    unijohab2html -i hangul-base.hex -o hangul-syllables.html\n\n");
00652
00653             exit (EXIT_SUCCESS);
00654         }
00655
00656         arg_count++;
00657     }
00658
00659     return;
00660 }

```

Here is the caller graph for this function:

5.41 unijohab2html.c

[Go to the documentation of this file.](#)

```

00001 /**
00002  @file unijohab2html.c
00003
00004  @brief Display overlapped Hangul letter combinations in a grid.
00005
00006  This displays overlapped letters that form Unicode Hangul Syllables
00007  combinations, as a tool to determine bounding boxes for all combinations.
00008  It works with both modern and archaic Hangul letters.
00009
00010  Input is a Unifont .hex file such as the "hangul-base.hex" file that
00011  is part of the Unifont package. Glyphs are all processed as being
00012  16 pixels wide and 16 pixels tall.
00013
00014  Output is an HTML file containing 16 by 16 pixel grids shwoing
00015  overlaps in table format, arranged by variation of the initial
00016  consonant (choseong).
00017
00018  Initial consonants (choseong) have 6 variations. In general, the
00019  first three are for combining with vowels (jungseong) that are
00020  vertical, horizontal, or vertical and horizontal, respectively;
00021  the second set of three variations are for combinations with a final

```

```

00022 consonant.
00023
00024 The output HTML file can be viewed in a web browser.
00025
00026 @author Paul Hardy
00027
00028 @copyright Copyright © 2023 Paul Hardy
00029 */
00030 /*
00031 LICENSE:
00032
00033 This program is free software: you can redistribute it and/or modify
00034 it under the terms of the GNU General Public License as published by
00035 the Free Software Foundation, either version 2 of the License, or
00036 (at your option) any later version.
00037
00038 This program is distributed in the hope that it will be useful,
00039 but WITHOUT ANY WARRANTY; without even the implied warranty of
00040 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00041 GNU General Public License for more details.
00042
00043 You should have received a copy of the GNU General Public License
00044 along with this program. If not, see <http://www.gnu.org/licenses/>.
00045 */
00046 #include <stdio.h>
00047 #include <stdlib.h>
00048 #include <string.h>
00049 #include "hangul.h"
00050
00051 #define MAXFILENAME 1024
00052
00053 #define START_JUNG 0 ///< Vowel index of first vowel with which to begin.
00054 // #define START_JUNG 21 /* Use this #define for just ancient vowels */
00055
00056
00057 /* (Red, Green, Blue) HTML color coordinates. */
00058 #define RED 0xCC0000 ///< Color code for slightly unsaturated HTML red.
00059 #define GREEN 0x00CC00 ///< Color code for slightly unsaturated HTML green.
00060 #define BLUE 0x0000CC ///< Color code for slightly unsaturated HTML blue.
00061 #define BLACK 0x000000 ///< Color code for HTML black.
00062 #define WHITE 0xFFFFFF ///< Color code for HTML white.
00063
00064
00065
00066 /**
00067 @brief The main function.
00068 */
00069 int
00070 main (int argc, char *argv[]) {
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
00099     variations, there will be no overlap. Then go through
00100     choseong, and then jongseong to find the overlapping
00101     combinations. This saves storage space as an alternative
00102     to storing large 2- or 3-dimensional overlap matrices.

```

```

00103  */
00104  // jungcho: Jungseong overlap with Choseong
00105  unsigned jungcho [TOTAL_JUNG * JUNG_VARIATIONS];
00106  // jongjung: Jongseong overlap with Jungseong -- for future expansion
00107  // unsigned jongjung [TOTAL_JUNG * JUNG_VARIATIONS];
00108
00109  int glyphs_overlap; /* If glyph pair being considered overlap. */
00110  int cho_overlaps = 0; /* Number of choseong+vowel overlaps. */
00111  // int jongjung_overlaps = 0; /* Number of vowel+jongseong overlaps. */
00112
00113  int inindex = 0;
00114  int outindex = 0;
00115  FILE *infp, *outfp; /* Input and output file pointers. */
00116
00117  void parse_args (int argc, char *argv[], int *inindex, int *outindex,
00118                  int *modern_only);
00119  int cho_variation (int cho, int jung, int jong);
00120  unsigned hangul_read_base16 (FILE *infp, unsigned glyph[][16]);
00121  int glyph_overlap (unsigned *glyph1, unsigned *glyph2);
00122
00123  void combine_glyphs (unsigned *glyph1, unsigned *glyph2,
00124                      unsigned *combined_glyph);
00125  void print_glyph_txt (FILE *fp, unsigned codept, unsigned *this_glyph);
00126
00127  /*
00128   Parse command line arguments to open input & output files, if given.
00129  */
00130  if (argc > 1) {
00131      parse_args (argc, argv, &inindex, &outindex, &modern_only);
00132  }
00133
00134  if (inindex == 0) {
00135      infp = stdin;
00136  }
00137  else {
00138      infp = fopen (argv[inindex], "r");
00139      if (infp == NULL) {
00140          fprintf (stderr, "\n*** ERROR: Cannot open %s for input.\n\n",
00141                  argv[inindex]);
00142          exit (EXIT_FAILURE);
00143      }
00144  }
00145
00146  if (outindex == 0) {
00147      outfp = stdout;
00148  }
00149  else {
00150      outfp = fopen (argv[outindex], "w");
00151      if (outfp == NULL) {
00152          fprintf (stderr, "\n*** ERROR: Cannot open %s for output.\n\n",
00153                  argv[outindex]);
00154          exit (EXIT_FAILURE);
00155      }
00156  }
00157
00158  /*
00159   Initialize glyph array to all zeroes.
00160  */
00161  for (codept = 0; codept < MAX_GLYPHS; codept++) {
00162      for (i = 0; i < 16; i++) glyph[codept][i] = 0x0000;
00163  }
00164
00165  /*
00166   Initialize overlap matrices to all zeroes.
00167  */
00168  for (i = 0; i < TOTAL_JUNG * JUNG_VARIATIONS; i++) {
00169      jungcho [i] = 0;
00170  }
00171  // jongjung is reserved for expansion.
00172  // for (i = 0; i < TOTAL_JONG * JONG_VARIATIONS; i++) {
00173  //     jongjung [i] = 0;
00174  // }
00175
00176  /*
00177   Read Hangul base glyph file.
00178  */
00179  max_codept = hangul_read_base16 (infp, glyph);
00180  if (max_codept > 0x8FFF) {
00181      fprintf (stderr, "\nWARNING: Hangul glyph range exceeds PUA space.\n\n");
00182  }
00183

```

```

00184  /*
00185     If only examining modern Hangul, fill the ancient glyphs
00186     with blanks to guarantee they won't overlap. This is
00187     not as efficient as ending loops sooner, but is easier
00188     to verify for correctness.
00189  */
00190  if (modern_only) {
00191      for (i = 0x0073; i < JUNG_HEX; i++) {
00192          for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
00193      }
00194      for (i = 0x027A; i < JONG_HEX; i++) {
00195          for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
00196      }
00197      for (i = 0x032B; i < 0x0400; i++) {
00198          for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
00199      }
00200  }
00201
00202  /*
00203     Initialize grids to all black (no color) for each of
00204     the 12 Choseong groups.
00205  */
00206  for (group = 0; group < 12; group++) {
00207      for (i = 0; i < 16; i++) {
00208          for (j = 0; j < 16; j++) {
00209              grid[group][i][j] = BLACK; /* No color at first */
00210          }
00211      }
00212  }
00213
00214  /*
00215     Superimpose all Choseong glyphs according to group.
00216     Each grid spot with choseong will be blue.
00217  */
00218  for (group = 0; group < 6; group++) {
00219      for (consonant1 = CHO_HEX + group;
00220           consonant1 < CHO_HEX +
00221                 CHO_VARIATIONS * TOTAL_CHO;
00222           consonant1 += CHO_VARIATIONS) {
00223          for (i = 0; i < 16; i++) { /* For each glyph row */
00224              mask = 0x8000;
00225              for (j = 0; j < 16; j++) {
00226                  if (glyph[consonant1][i] & mask) grid[group][i][j] |= BLUE;
00227                  mask >>= 1; /* Get next bit in glyph row */
00228              }
00229          }
00230      }
00231  }
00232
00233  /*
00234     Fill with Choseong (initial consonant) to prepare
00235     for groups 3-5 with jongseong except niuen (group+3),
00236     then for groups 3-5 with jongseong nieun (group+6).
00237  */
00238  for (group = 3; group < 6; group++) {
00239      for (i = 0; i < 16; i++) {
00240          for (j = 0; j < 16; j++) {
00241              grid[group + 6][i][j] = grid[group + 3][i][j]
00242                               = grid[group][i][j];
00243          }
00244      }
00245  }
00246
00247  /*
00248     For each Jungseong, superimpose first variation on
00249     appropriate Choseong group for grids 0 to 5.
00250  */
00251  for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
00252      group = cho_variation (-1, vowel, -1);
00253      glyphs_overlap = 0; /* Assume the 2 glyphs do not overlap. */
00254
00255      for (i = 0; i < 16; i++) { /* For each glyph row */
00256          mask = 0x8000;
00257          for (j = 0; j < 16; j++) {
00258              if (glyph[JUNG_HEX + JUNG_VARIATIONS * vowel][i] & mask) {
00259                  /*
00260                     If there was already blue in this grid cell,
00261                     mark this vowel variation as having overlap
00262                     with choseong (initial consonant) letter(s).
00263                  */
00264                  if (grid[group][i][j] & BLUE) glyphs_overlap = 1;

```



```

00265
00266     /* Add green to grid cell color. */
00267     grid[group][i][j] |= GREEN;
00268 }
00269     mask »= 1; /* Mask for next bit in glyph row */
00270 } /* for j */
00271 } /* for i */
00272 if (glyphs_overlap) {
00273     jungcho [JUNG_VARIATIONS * vowel] = 1;
00274     cho_overlaps++;
00275 }
00276 } /* for each vowel */
00277
00278 /*
00279  For each Jungseong, superimpose second variation on
00280  appropriate Choseong group for grids 6 to 8.
00281 */
00282 for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
00283     /*
00284      The second vowel variation is for combination with
00285      a final consonant (Jongseong), with initial consonant
00286      (Choseong) variations (or "groups") 3 to 5. Thus,
00287      if the vowel type returns an initial Choseong group
00288      of 0 to 2, add 3 to it.
00289     */
00290     group = cho_variation (-1, vowel, -1);
00291     /*
00292      Groups 0 to 2 don't use second vowel variation,
00293      so increment if group is below 2.
00294     */
00295     if (group < 3) group += 3;
00296     glyphs_overlap = 0; /* Assume the 2 glyphs do not overlap. */
00297
00298     for (i = 0; i < 16; i++) { /* For each glyph row */
00299         mask = 0x8000; /* Start mask at leftmost glyph bit */
00300         for (j = 0; j < 16; j++) { /* For each column in this row */
00301             /* " + 1" is to get each vowel's second variation */
00302             if (glyph [JUNG_HEX +
00303                     JUNG_VARIATIONS * vowel + 1][i] & mask) {
00304                 /* If this cell has blue already, mark as overlapped. */
00305                 if (grid [group + 3][i][j] & BLUE) glyphs_overlap = 1;
00306             }
00307             /* Superimpose green on current cell color. */
00308             grid [group + 3][i][j] |= GREEN;
00309         }
00310         mask »= 1; /* Get next bit in glyph row */
00311     } /* for j */
00312 } /* for i */
00313 if (glyphs_overlap) {
00314     jungcho [JUNG_VARIATIONS * vowel + 1] = 1;
00315     cho_overlaps++;
00316 }
00317 } /* for each vowel */
00318
00319 /*
00320  For each Jungseong, superimpose third variation on
00321  appropriate Choseong group for grids 9 to 11 for
00322  final consonant (Jongseong) of Nieun.
00323 */
00324 for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
00325     group = cho_variation (-1, vowel, -1);
00326     if (group < 3) group += 3;
00327     glyphs_overlap = 0; /* Assume the 2 glyphs do not overlap. */
00328
00329     for (i = 0; i < 16; i++) { /* For each glyph row */
00330         mask = 0x8000;
00331         for (j = 0; j < 16; j++) {
00332             if (glyph [JUNG_HEX +
00333                     JUNG_VARIATIONS * vowel + 2][i] & mask) {
00334                 /* If this cell has blue already, mark as overlapped. */
00335                 if (grid [group + 6][i][j] & BLUE) glyphs_overlap = 1;
00336             }
00337             grid [group + 6][i][j] |= GREEN;
00338         }
00339         mask »= 1; /* Get next bit in glyph row */
00340     } /* for j */
00341 } /* for i */
00342 if (glyphs_overlap) {
00343     jungcho [JUNG_VARIATIONS * vowel + 2] = 1;
00344     cho_overlaps++;
00345 }

```

```

00346 } /* for each vowel */
00347
00348
00349 /*
00350 Superimpose all final consonants except nieun for grids 6 to 8.
00351 */
00352 for (consonant2 = 0; consonant2 < TOTAL__JONG; consonant2++) {
00353     /*
00354     Skip over Jongseong Nieun, because it is covered in
00355     grids 9 to 11 after this loop.
00356     */
00357     if (consonant2 == 3) consonant2++;
00358
00359     glyphs_overlap = 0; /* Assume the 2 glyphs do not overlap. */
00360     for (i = 0; i < 16; i++) { /* For each glyph row */
00361         mask = 0x8000;
00362         for (j = 0; j < 16; j++) {
00363             if (glyph [JONG_HEX +
00364                 JONG_VARIATIONS * consonant2][i] & mask) {
00365                 if (grid[6][i][j] & GREEN ||
00366                     grid[7][i][j] & GREEN ||
00367                     grid[8][i][j] & GREEN) glyphs_overlap = 1;
00368
00369                 grid[6][i][j] |= RED;
00370                 grid[7][i][j] |= RED;
00371                 grid[8][i][j] |= RED;
00372             }
00373             mask »= 1; /* Get next bit in glyph row */
00374         } /* for j */
00375     } /* for i */
00376     // jongjung is for expansion
00377     // if (glyphs_overlap) {
00378     //     jongjung [JONG_VARIATIONS * consonant2] = 1;
00379     //     jongjung_overlaps++;
00380     // }
00381 } /* for each final consonant except nieun */
00382
00383 /*
00384 Superimpose final consonant 3 (Jongseong Nieun) on
00385 groups 9 to 11.
00386 */
00387 codept = JONG_HEX + 3 * JONG_VARIATIONS;
00388
00389 for (i = 0; i < 16; i++) { /* For each glyph row */
00390     mask = 0x8000;
00391     for (j = 0; j < 16; j++) {
00392         if (glyph[codept][i] & mask) {
00393             grid[9][i][j] |= RED;
00394             grid[10][i][j] |= RED;
00395             grid[11][i][j] |= RED;
00396         }
00397         mask »= 1; /* Get next bit in glyph row */
00398     }
00399 }
00400
00401
00402 /*
00403 Turn the black (uncolored) cells into white for better
00404 visibility of grid when displayed.
00405 */
00406 for (group = 0; group < 12; group++) {
00407     for (i = 0; i < 16; i++) {
00408         for (j = 0; j < 16; j++) {
00409             if (grid[group][i][j] == BLACK) grid[group][i][j] = WHITE;
00410         }
00411     }
00412 }
00413
00414
00415 /*
00416 Generate HTML output.
00417 */
00418 fprintf (outfp, "<html>\n");
00419 fprintf (outfp, "<head>\n");
00420 fprintf (outfp, "    <title>Johab 6/3/1 Overlaps</title>\n");
00421 fprintf (outfp, "</head>\n");
00422 fprintf (outfp, "<body bgcolor=\\"#FFFFCC\">\n");
00423
00424 fprintf (outfp, "<center>\n");
00425 fprintf (outfp, "    <h1>Unifont Hangul Jamo Syllable Components</h1>\n");
00426 fprintf (outfp, "    <h2>Johab 6/3/1 Overlap</h2><br><br>\n");

```

```

00427
00428 /* Print the color code key for the table. */
00429 fprintf (outfp, " <table border=\"1\" cellpadding=\"10\">\n");
00430 fprintf (outfp, " <tr><th colspan=\"2\" align=\"center\" bgcolor=\"#FFCC80\">");
00431 fprintf (outfp, "<font size=\"+1\">Key</font></th></tr>\n");
00432 fprintf (outfp, " <tr>\n");
00433 fprintf (outfp, " <th align=\"center\" bgcolor=\"#FFFF80\">Color</th>\n");
00434 fprintf (outfp, " <th align=\"center\" bgcolor=\"#FFFF80\">Letter(s)</th>\n");
00435 fprintf (outfp, " </tr>\n");
00436
00437 fprintf (outfp, " <tr><td bgcolor=\"#06X\">", BLUE);
00438 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00439 fprintf (outfp, "<td>Choseong (Initial Consonant)</td></tr>\n");
00440
00441 fprintf (outfp, " <tr><td bgcolor=\"#06X\">", GREEN);
00442 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00443 fprintf (outfp, "<td>Jungseong (Medial Vowel/Diphthong)</td></tr>\n");
00444
00445 fprintf (outfp, " <tr><td bgcolor=\"#06X\">", RED);
00446 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00447 fprintf (outfp, "<td>Jongseong (Final Consonant)</td></tr>\n");
00448
00449 fprintf (outfp, " <tr><td bgcolor=\"#06X\">", BLUE | GREEN);
00450 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00451 fprintf (outfp, "<td>Choseong + Jungseong Overlap</td></tr>\n");
00452
00453 fprintf (outfp, " <tr><td bgcolor=\"#06X\">", GREEN | RED);
00454 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00455 fprintf (outfp, "<td>Jungseong + Jongseong Overlap</td></tr>\n");
00456
00457 fprintf (outfp, " <tr><td bgcolor=\"#06X\">", RED | BLUE);
00458 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00459 fprintf (outfp, "<td>Choseong + Jongseong Overlap</td></tr>\n");
00460
00461 fprintf (outfp, " <tr><td bgcolor=\"#06X\">", RED | GREEN | BLUE);
00462 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>");
00463 fprintf (outfp, "<td>Choseong + Jungseong + Jongseong Overlap</td></tr>\n");
00464
00465 fprintf (outfp, " </table>\n");
00466 fprintf (outfp, " <br><br>\n");
00467
00468
00469 for (group = 0; group < 12; group++) {
00470 /* Arrange tables 3 across, 3 down. */
00471 if ((group % 3) == 0) {
00472 fprintf (outfp, " <table border=\"0\" cellpadding=\"10\">\n");
00473 fprintf (outfp, " <tr>\n");
00474 }
00475
00476 fprintf (outfp, " <td>\n");
00477 fprintf (outfp, " <table border=\"3\" cellpadding=\"2\">\n");
00478 fprintf (outfp, " <tr><th colspan=\"16\" bgcolor=\"#FFFF80\">");
00479 fprintf (outfp, "Choseong Group %d, %s %s</th></tr>\n",
00480 group < 6 ? group : (group > 8 ? group - 6 : group - 3),
00481 group < 6 ? (group < 3 ? "No" : "Without") : "With",
00482 group < 9 ? "Jongseong" : "Nieun");
00483
00484 for (i = 0; i < 16; i++) {
00485 fprintf (outfp, " <tr>\n");
00486 for (j = 0; j < 16; j++) {
00487 fprintf (outfp, " <td bgcolor=\"#06X\">",
00488 grid[group][i][j]);
00489 fprintf (outfp, "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>\n");
00490 }
00491 fprintf (outfp, " </tr>\n");
00492 }
00493
00494 fprintf (outfp, " </td>\n");
00495 fprintf (outfp, " </tr>\n");
00496 fprintf (outfp, " </table>\n");
00497 fprintf (outfp, " </td>\n");
00498
00499 if ((group % 3) == 2) {
00500 fprintf (outfp, " </tr>\n");
00501 fprintf (outfp, " </table>\n <br>\n");
00502 }
00503 }
00504
00505 /* Wrap up HTML table output. */
00506 fprintf (outfp, " </center>\n");
00507

```

```

00508  /*
00509  Print overlapping initial consonant + vowel combinations.
00510  */
00511  fprintf (outfp, "<h2>%d Vowel Overlaps with Initial Consonants Found</h2>",
00512           cho_overlaps);
00513  fprintf (outfp, "<font size=\")+1\"><pre>\n");
00514
00515  for (i = JUNG_HEX;
00516       i < JUNG_HEX + TOTAL_JUNG * JUNG_VARIATIONS;
00517       i++) {
00518      /*
00519       If this vowel variation (Jungseong) had overlaps
00520       with one or more initial consonants (Choseong),
00521       find and print them.
00522      */
00523      if (jungcho [i - JUNG_HEX]) {
00524          ancient_choseong = 0; /* Not within ancient choseong range yet. */
00525          fprintf (outfp, "<font color=\")+1\"><b>");
00526          if (i >= JUNG_ANCIENT_HEX) {
00527              if (i >= JUNG_EXTB_HEX) fprintf (outfp, "Extended-B ");
00528              fprintf (outfp, "Ancient ");
00529          }
00530          fprintf (outfp, "Vowel at 0x%04X and&hellip;</b>", i + PUA_START);
00531          fprintf (outfp, "</font>\n\n");
00532
00533          /*
00534           Get current vowel number, 0 to (TOTAL_JUNG - 1), and
00535           current vowel variation, 0 or 1, or 2 for final nieun.
00536          */
00537          vowel = (i - JUNG_HEX) / JUNG_VARIATIONS;
00538          vowel_variation = (i - JUNG_HEX) % JUNG_VARIATIONS;
00539
00540          /* Get first Choseong group for this vowel, 0 to 5. */
00541          group = cho_variation (-1, vowel, -1);
00542
00543          /*
00544           If this vowel variation is used with a final consonant
00545           (Jongseong) and the default initial consonant (Choseong)
00546           group for this vowel is < 3, add 3 to current Chosenong
00547           group.
00548          */
00549          if (vowel_variation > 0 && group < 3) group += 3;
00550
00551          for (consonant1 = 0; consonant1 < TOTAL_CHO; consonant1++) {
00552              overlapped = glyph_overlap (glyph [i],
00553                                         glyph [consonant1 * CHO_VARIATIONS
00554                                                  + CHO_HEX + group]);
00555
00556              /*
00557               If we just entered ancient choseong range, flag it.
00558              */
00559              if (overlapped && consonant1 >= 19 && ancient_choseong == 0) {
00560                  fprintf (outfp, "<font color=\")+1\"><b>");
00561                  fprintf (outfp, "&hellip;Ancient Choseong&hellip;</b></font>\n");
00562                  ancient_choseong = 1;
00563              }
00564              /*
00565               If overlapping choseong found, print combined glyph.
00566              */
00567              if (overlapped != 0) {
00568
00569                  combine_glyphs (glyph [i],
00570                                glyph [consonant1 * CHO_VARIATIONS
00571                                       + CHO_HEX + group],
00572                                tmp_glyph);
00573
00574                  print_glyph_txt (outfp,
00575                                  PUA_START +
00576                                  consonant1 * CHO_VARIATIONS +
00577                                  CHO_HEX + group,
00578                                  tmp_glyph);
00579
00580              } /* If overlapping pixels found. */
00581          } /* For each initial consonant (Choseong) */
00582      } /* Find the initial consonant that overlapped this vowel variation. */
00583  } /* For each variation of each vowel (Jungseong) */
00584
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 }
00597
00598
00599 /**
00600  @brief Parse command line arguments.
00601
00602  @param[in] argc The argc parameter to the main function.
00603  @param[in] argv The argv command line arguments to the main function.
00604  @param[in,out] infile The input filename; defaults to NULL.
00605  @param[in,out] outfile The output filename; defaults to NULL.
00606 */
00607 void
00608 parse_args (int argc, char *argv[], int *inindex, int *outindex,
00609             int *modern_only) {
00610     int arg_count; /* Current index into argv. */
00611
00612     int strcmp (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 (strcmp (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 (strcmp (argv [arg_count], "-m", 2) == 0 ||
00627                 strcmp (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 (strcmp (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 (strcmp (argv [arg_count], "-h", 2) == 0 ||
00639                 strcmp (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");
00647             printf ("    -i          input_file  Unifont hangul-base.hex formatted input file.\n\n");
00648             printf ("    -o          output_file  HTML output file showing overlapping letters.\n\n");
00649             printf ("    -m, --modern        Only examine modern Hangul letters.\n\n");
00650             printf ("    Example:\n\n");
00651             printf ("    unijohab2html -i hangul-base.hex -o hangul-syllables.html\n\n");
00652
00653             exit (EXIT_SUCCESS);
00654         }
00655
00656         arg_count++;
00657     }
00658
00659     return;
00660 }
00661

```

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

Copyright (C) 2007, 2008, 2013, 2014 Paul Hardy

This program counts the number of glyphs that are defined in each "page" of 256 code points, and prints the counts in an 8 x 8 grid. Input is from stdin. Output is to stdout.

The background color of each cell in a 16-by-16 grid of 256 code points is shaded to indicate percentage coverage. Red indicates 0% coverage, green represents 100% coverage, and colors in between pure red and pure green indicate partial coverage on a scale.

Each code point range number can be a hyperlink to a PNG file for that 256-code point range's corresponding bitmap glyph image.

Synopsis:

```
unipagecount < font_file.hex > count.txt
unipagecount -phex_page_num < font_file.hex -- just 256 points
unipagecount -h < font_file.hex -- HTML table
unipagecount -P1 -h < font.hex > count.html -- Plane 1, HTML out
unipagecount -l < font_file.hex -- linked HTML table
```

Definition in file [unipagecount.c](#).

5.42.2 Macro Definition Documentation

5.42.2.1 MAXBUF

```
#define MAXBUF 256
Maximum input line size - 1.
Definition at line 63 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 74 of file [unipagecount.c](#).

```
00075 {
00076
00077     char inbuf[MAXBUF]; /* Max 256 characters in an input line */
00078     int i, j; /* loop variables */
00079     unsigned plane=0; /* Unicode plane number, 0 to 0x16 */
00080     unsigned page; /* unicode page (256 bytes wide) */
00081     unsigned uchar; /* unicode character */
00082     int pagecount[256] = {256 * 0};
00083     int onepage=0; /* set to one if printing character grid for one page */
00084     unsigned pageno=0; /* page number selected if only examining one page */
00085     int html=0; /* =0: print plain text; =1: print HTML */
00086     int links=0; /* =1: print HTML links; =0: don't print links */
00087
00088     /* make (print) flipped HTML table */
00089     void mkftable (unsigned plane, int pagecount[256], int links);
00090
00091     size_t strlen(const char *s);
00092
00093     if (argc > 1 && argv[1][0] == '-') { /* Parse option */
00094         plane = 0;
00095         for (i = 1; i < argc; i++) {
00096             switch (argv[i][1]) {
00097                 case 'p': /* specified -p<hexpage> -- use given page number */
00098                     sscanf (&argv[i][2], "%x", &pageno);
00099                     if (pageno >= 0 && pageno <= 255) onepage = 1;
00100                     break;
00101                 case 'h': /* print HTML table instead of text table */
00102                     html = 1;
00103                     break;
00104                 case 'l': /* print hyperlinks in HTML table */
00105                     links = 1;
00106                     html = 1;
00107                     break;
00108                 case 'P': /* Plane number specified */
00109                     plane = atoi(&argv[i][2]);
00110                     break;
00111             }
00112         }
00113     }
00114     /*
00115      * Initialize pagecount to account for noncharacters.
00116      */
00117     if (!onepage && plane==0) {
00118         pagecount[0xfd] = 32; /* for U+FDD0..U+FDEF */
00119     }
00120     pagecount[0xff] = 2; /* for U+nnFFFE, U+nnFFFF */
00121     /*
00122      * Read one line at a time from input. The format is:
00123
00124         <hexpos>:<hexbitmap>
00125
00126      * where <hexpos> is the hexadecimal Unicode character position
00127      * in the range 00..FF and <hexbitmap> is the sequence of hexadecimal
00128      * digits of the character, laid out in a grid from left to right,
00129      * top to bottom. The character is assumed to be 16 rows of variable
00130      * width.
00131      */
```

```

00132 while (fgets (inbuf, MAXBUF-1, stdin) != NULL) {
00133     sscanf (inbuf, "%X", &unichar);
00134     page = unichar » 8;
00135     if (onepage) { /* only increment counter if this is page we want */
00136         if (page == pageno) { /* character is in the page we want */
00137             pagecount[unichar & 0xff]++; /* mark character as covered */
00138         }
00139     }
00140     else { /* counting all characters in all pages */
00141         if (plane == 0) {
00142             /* Don't add in noncharacters (U+FDD0..U+FDEF, U+FFFE, U+FFFF) */
00143             if (unichar < 0xfdd0 || (unichar > 0xfdef && unichar < 0xfffe))
00144                 pagecount[page]++;
00145         }
00146         else {
00147             if ((page » 8) == plane) { /* code point is in desired plane */
00148                 pagecount[page & 0xFF]++;
00149             }
00150         }
00151     }
00152 }
00153 if (html) {
00154     mkftable (plane, pagecount, links);
00155 }
00156 else { /* Otherwise, print plain text table */
00157     if (plane > 0) fprintf (stdout, " ");
00158     fprintf (stdout,
00159         " 0 1 2 3 4 5 6 7 8 9 A B C D E F\n");
00160     for (i=0; i<0x10; i++) {
00161         fprintf (stdout, "%02X%X ", plane, i); /* row header */
00162         for (j=0; j<0x10; j++) {
00163             if (onepage) {
00164                 if (pagecount[i*16+j])
00165                     fprintf (stdout, " * ");
00166             }
00167             else {
00168                 fprintf (stdout, " . ");
00169             }
00170             else {
00171                 fprintf (stdout, "%3X ", pagecount[i*16+j]);
00172             }
00173         }
00174         fprintf (stdout, "\n");
00175     }
00176 }
00177 exit (0);
00178 }

```

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 194 of file [unipagecount.c](#).

```

00195 {
00196     int i, j;
00197     int count;
00198     unsigned bgcolor;
00199 }

```



```

00200 printf("<html>\n");
00201 printf("<body>\n");
00202 printf("<table border=\"3\" align=\"center\">\n");
00203 printf(" <tr><th colspan=\"16\" bgcolor=\"#ffcc80\">");
00204 printf("GNU Unifont Glyphs<br>with Page Coverage for Plane %d<br>(Green=100%%, Red=0%%)</th></tr>\n",
plane);
00205 for (i = 0x0; i <= 0xF; i++) {
00206     printf(" <tr>\n");
00207     for (j = 0x0; j <= 0xF; j++) {
00208         count = pagecount[ (i « 4) | j ];
00209
00210         /* print link in cell if links == 1 */
00211         if (plane != 0 || (i < 0xd || (i == 0xd && j < 0x8) || (i == 0xf && j > 0x8))) {
00212             /* background color is light green if completely done */
00213             if (count == 0x100) bgcolor = 0xccffcc;
00214             /* otherwise background is a shade of yellow to orange to red */
00215             else bgcolor = 0xff0000 | (count « 8) | (count » 1);
00216             printf(" <td bgcolor=\"%06X\">", bgcolor);
00217             if (plane == 0)
00218                 printf("<a href=\"png/plane%02X/uni%02X%X%X.png\">%X%X</a>", plane, plane, i, j, j);
00219             else
00220                 printf("<a href=\"png/plane%02X/uni%02X%X%X.png\">%02X%X%X</a>", plane, plane, i, j, plane, i, j);
00221             printf("</td>\n");
00222         }
00223         else if (i == 0xd) {
00224             if (j == 0x8) {
00225                 printf(" <td align=\"center\" colspan=\"8\" bgcolor=\"#cccccc\">");
00226                 printf("<b>Surrogate Pairs</b>");
00227                 printf("</td>\n");
00228             } /* otherwise don't print anything more columns in this row */
00229         }
00230         else if (i == 0xe) {
00231             if (j == 0x0) {
00232                 printf(" <td align=\"center\" colspan=\"16\" bgcolor=\"#cccccc\">");
00233                 printf("<b>Private Use Area</b>");
00234                 printf("</td>\n");
00235             } /* otherwise don't print any more columns in this row */
00236         }
00237         else if (i == 0xf) {
00238             if (j == 0x0) {
00239                 printf(" <td align=\"center\" colspan=\"9\" bgcolor=\"#cccccc\">");
00240                 printf("<b>Private Use Area</b>");
00241                 printf("</td>\n");
00242             }
00243         }
00244     }
00245     printf(" </tr>\n");
00246 }
00247 printf("</table>\n");
00248 printf("</body>\n");
00249 printf("</html>\n");
00250
00251 return;
00252 }

```

Here is the caller graph for this function:

5.43 unipagecount.c

[Go to the documentation of this file.](#)

```

00001 /**
00002  @file unipagecount.c
00003
00004  @brief unipagecount - Count the number of glyphs defined in each page
00005         of 256 code points
00006
00007  @author Paul Hardy, unifoundry <at> unifoundry.com, December 2007
00008
00009  @copyright Copyright (C) 2007, 2008, 2013, 2014 Paul Hardy
00010
00011  This program counts the number of glyphs that are defined in each
00012  "page" of 256 code points, and prints the counts in an 8 x 8 grid.
00013  Input is from stdin. Output is to stdout.
00014
00015  The background color of each cell in a 16-by-16 grid of 256 code points
00016  is shaded to indicate percentage coverage. Red indicates 0% coverage,
00017  green represents 100% coverage, and colors in between pure red and pure
00018  green indicate partial coverage on a scale.
00019

```

```

00020 Each code point range number can be a hyperlink to a PNG file for
00021 that 256-code point range's corresponding bitmap glyph image.
00022
00023 Synopsis:
00024
00025     unipagecount < font_file.hex > count.txt
00026     unipagecount -phex_page_num < font_file.hex -- just 256 points
00027     unipagecount -h < font_file.hex -- HTML table
00028     unipagecount -P1 -h < font.hex > count.html -- Plane 1, HTML out
00029     unipagecount -l < font_file.hex -- linked HTML table
00030 */
00031 /*
00032 LICENSE:
00033
00034     This program is free software: you can redistribute it and/or modify
00035     it under the terms of the GNU General Public License as published by
00036     the Free Software Foundation, either version 2 of the License, or
00037     (at your option) any later version.
00038
00039     This program is distributed in the hope that it will be useful,
00040     but WITHOUT ANY WARRANTY; without even the implied warranty of
00041     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00042     GNU General Public License for more details.
00043
00044     You should have received a copy of the GNU General Public License
00045     along with this program. If not, see <http://www.gnu.org/licenses/>.
00046 */
00047
00048 /*
00049     2018, Paul Hardy: Changed "Private Use" to "Private Use Area" in
00050     output HTML file.
00051
00052     21 October 2023 [Paul Hardy]:
00053     - Added full prototype for mkftable function in main function.
00054
00055     6 September 2025 [Paul Hardy]:
00056     - Changed pageno from "int" to "unsigned" for compatibility
00057     with sscanf definition.
00058 */
00059
00060 #include <stdio.h>
00061 #include <stdlib.h>
00062
00063 #define MAXBUF 256 ///< Maximum input line size - 1.
00064
00065 /**
00066  * @brief The main function.
00067  *
00068  * @param[in] argc The count of command line arguments.
00069  * @param[in] argv Pointer to array of command line arguments.
00070  * @return This program exits with status 0.
00071  */
00072 */
00073 int
00074 main (int argc, char *argv[])
00075 {
00076     char inbuf[MAXBUF]; /* Max 256 characters in an input line */
00077     int i, j; /* loop variables */
00078     unsigned plane=0; /* Unicode plane number, 0 to 0x16 */
00079     unsigned page; /* unicode page (256 bytes wide) */
00080     unsigned unichar; /* unicode character */
00081     int pagecount[256] = {256 * 0};
00082     int onepage=0; /* set to one if printing character grid for one page */
00083     unsigned pageno=0; /* page number selected if only examining one page */
00084     int html=0; /* =0: print plain text; =1: print HTML */
00085     int links=0; /* =1: print HTML links; =0: don't print links */
00086
00087     /* make (print) flipped HTML table */
00088     void mkftable (unsigned plane, int pagecount[256], int links);
00089
00090     size_t strlen(const char *s);
00091
00092     if (argc > 1 && argv[1][0] == '-') { /* Parse option */
00093         plane = 0;
00094         for (i = 1; i < argc; i++) {
00095             switch (argv[i][1]) {
00096                 case 'p': /* specified -p<hexpage> -- use given page number */
00097                     sscanf (&argv[i][2], "%x", &pageno);
00098                     if (pageno >= 0 && pageno <= 255) onepage = 1;
00099                     break;
00100

```

```

00101         case 'h': /* print HTML table instead of text table */
00102             html = 1;
00103             break;
00104         case 'l': /* print hyperlinks in HTML table */
00105             links = 1;
00106             html = 1;
00107             break;
00108         case 'P': /* Plane number specified */
00109             plane = atoi(&argv[1][2]);
00110             break;
00111     }
00112 }
00113 }
00114 /*
00115     Initialize pagecount to account for noncharacters.
00116 */
00117 if (!onepage && plane==0) {
00118     pagecount[0xfd] = 32; /* for U+FDD0..U+FDEF */
00119 }
00120 pagecount[0xff] = 2; /* for U+nnFFFE, U+nnFFFF */
00121 /*
00122     Read one line at a time from input. The format is:
00123
00124     <hexpos>:<hexbitmap>
00125
00126     where <hexpos> is the hexadecimal Unicode character position
00127     in the range 00..FF and <hexbitmap> is the sequence of hexadecimal
00128     digits of the character, laid out in a grid from left to right,
00129     top to bottom. The character is assumed to be 16 rows of variable
00130     width.
00131 */
00132 while (fgets (inbuf, MAXBUF-1, stdin) != NULL) {
00133     sscanf (inbuf, "%X", &unichar);
00134     page = unichar » 8;
00135     if (onepage) { /* only increment counter if this is page we want */
00136         if (page == pageno) { /* character is in the page we want */
00137             pagecount[unichar & 0xff]++; /* mark character as covered */
00138         }
00139     }
00140     else { /* counting all characters in all pages */
00141         if (plane == 0) {
00142             /* Don't add in noncharacters (U+FDD0..U+FDEF, U+FFFE, U+FFFF) */
00143             if (unichar < 0xfdd0 || (unichar > 0xfdef && unichar < 0xfffe))
00144                 pagecount[page]++;
00145         }
00146         else {
00147             if ((page » 8) == plane) { /* code point is in desired plane */
00148                 pagecount[page & 0xFF]++;
00149             }
00150         }
00151     }
00152 }
00153 if (html) {
00154     mkftable (plane, pagecount, links);
00155 }
00156 else { /* Otherwise, print plain text table */
00157     if (plane > 0) fprintf (stdout, " ");
00158     fprintf (stdout,
00159         "    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F\n");
00160     for (i=0; i<0x10; i++) {
00161         fprintf (stdout, "%02X%X ", plane, i); /* row header */
00162         for (j=0; j<0x10; j++) {
00163             if (onepage) {
00164                 if (pagecount[i*16+j])
00165                     fprintf (stdout, "* ");
00166                 else
00167                     fprintf (stdout, ". ");
00168             }
00169             else {
00170                 fprintf (stdout, "%3X ", pagecount[i*16+j]);
00171             }
00172         }
00173         fprintf (stdout, "\n");
00174     }
00175 }
00176 }
00177 exit (0);
00178 }
00179
00180
00181 /**

```

```

00182  @brief Create an HTML table linked to PNG images.
00183
00184  This function creates an HTML table to show PNG files
00185  in a 16 by 16 grid. The background color of each "page"
00186  of 256 code points is shaded from red (for 0% coverage)
00187  to green (for 100% coverage).
00188
00189  @param[in] plane The Unicode plane, 0..17.
00190  @param[in] pagecount Array with count of glyphs in each 256 code point range.
00191  @param[in] links 1 = generate hyperlinks, 0 = do not generate hyperlinks.
00192  */
00193 void
00194 mkftable (unsigned plane, int pagecount[256], int links)
00195 {
00196     int i, j;
00197     int count;
00198     unsigned bgcolor;
00199
00200     printf("<html>\n");
00201     printf("<body>\n");
00202     printf("<table border=\"3\" align=\"center\">\n");
00203     printf(" <tr><th colspan=\"16\" bgcolor=\"#ffcc80\">");
00204     printf("GNU Unifont Glyphs<br>with Page Coverage for Plane %d<br>(Green=100%%, Red=0%%)</th></tr>\n",
plane);
00205     for (i = 0x0; i <= 0xF; i++) {
00206         printf(" <tr>\n");
00207         for (j = 0x0; j <= 0xF; j++) {
00208             count = pagecount[ (i « 4) | j ];
00209
00210             /* print link in cell if links == 1 */
00211             if (plane != 0 || (i < 0xd || (i == 0xd && j < 0x8) || (i == 0xf && j > 0x8))) {
00212                 /* background color is light green if completely done */
00213                 if (count == 0x100) bgcolor = 0xccffcc;
00214                 /* otherwise background is a shade of yellow to orange to red */
00215                 else bgcolor = 0xff0000 | (count « 8) | (count » 1);
00216                 printf(" <td bgcolor=\"%06X\">", bgcolor);
00217                 if (plane == 0)
00218                     printf("<a href=\"png/plane%02X/uni%02X%X%X.png\">%X%X</a>", plane, plane, i, j, j);
00219                 else
00220                     printf("<a href=\"png/plane%02X/uni%02X%X%X.png\">%02X%X%X</a>", plane, plane, i, j, plane, i, j);
00221                 printf("</td>\n");
00222             }
00223             else if (i == 0xd) {
00224                 if (j == 0x8) {
00225                     printf(" <td align=\"center\" colspan=\"8\" bgcolor=\"#cccccc\">");
00226                     printf("<b>Surrogate Pairs</b>");
00227                     printf("</td>\n");
00228                 } /* otherwise don't print anything more columns in this row */
00229             }
00230             else if (i == 0xe) {
00231                 if (j == 0x0) {
00232                     printf(" <td align=\"center\" colspan=\"16\" bgcolor=\"#cccccc\">");
00233                     printf("<b>Private Use Area</b>");
00234                     printf("</td>\n");
00235                 } /* otherwise don't print any more columns in this row */
00236             }
00237             else if (i == 0xf) {
00238                 if (j == 0x0) {
00239                     printf(" <td align=\"center\" colspan=\"9\" bgcolor=\"#cccccc\">");
00240                     printf("<b>Private Use Area</b>");
00241                     printf("</td>\n");
00242                 }
00243             }
00244         }
00245         printf(" </tr>\n");
00246     }
00247     printf("</table>\n");
00248     printf("</body>\n");
00249     printf("</html>\n");
00250
00251     return;
00252 }

```