

GNU Unifont

Generated by Doxygen 1.9.3

1 GNU Unifont	1
1.1 GNU Unifont C Utilities	1
1.2 LICENSE	1
1.3 Introduction	1
1.4 The C Programs	2
1.5 Perl Scripts	6
2 Data Structure Index	11
2.1 Data Structures	11
3 File Index	13
3.1 File List	13
4 Data Structure Documentation	15
4.1 Buffer Struct Reference	15
4.1.1 Detailed Description	15
4.1.2 Field Documentation	15
4.1.2.1 begin	15
4.1.2.2 capacity	16
4.1.2.3 end	16
4.1.2.4 next	16
4.2 Font Struct Reference	16
4.2.1 Detailed Description	17
4.2.2 Field Documentation	17
4.2.2.1 glyphCount	17
4.2.2.2 glyphs	17
4.2.2.3 maxWidth	17
4.2.2.4 tables	18
4.3 Glyph Struct Reference	18
4.3.1 Detailed Description	18
4.3.2 Field Documentation	18
4.3.2.1 bitmap	19
4.3.2.2 byteCount	19
4.3.2.3 codePoint	19
4.3.2.4 combining	19
4.3.2.5 lsb	19
4.3.2.6 pos	20
4.4 NamePair Struct Reference	20
4.4.1 Detailed Description	20
4.4.2 Field Documentation	20
4.4.2.1 id	20

4.4.2.2 str	21
4.5 Options Struct Reference	21
4.5.1 Detailed Description	21
4.5.2 Field Documentation	21
4.5.2.1 bitmap	21
4.5.2.2 blankOutline	22
4.5.2.3 cff	22
4.5.2.4 gpos	22
4.5.2.5 gsub	22
4.5.2.6 hex	22
4.5.2.7 nameStrings	22
4.5.2.8 out	23
4.5.2.9 pos	23
4.5.2.10 truetype	23
4.6 Table Struct Reference	23
4.6.1 Detailed Description	24
4.6.2 Field Documentation	24
4.6.2.1 content	24
4.6.2.2 tag	24
4.7 TableRecord Struct Reference	24
4.7.1 Detailed Description	25
4.7.2 Field Documentation	25
4.7.2.1 checksum	25
4.7.2.2 length	25
4.7.2.3 offset	25
4.7.2.4 tag	25
5 File Documentation	27
5.1 src/hex2otf.c File Reference	27
5.1.1 Detailed Description	32
5.1.2 Macro Definition Documentation	32
5.1.2.1 addByte	32
5.1.2.2 ASCENDER	32
5.1.2.3 B0	33
5.1.2.4 B1	33
5.1.2.5 BX	33
5.1.2.6 defineStore	33
5.1.2.7 DESCENDER	34
5.1.2.8 FU	34
5.1.2.9 FUPEM	34

5.1.2.10 GLYPH_HEIGHT	34
5.1.2.11 GLYPH_MAX_BYTE_COUNT	34
5.1.2.12 GLYPH_MAX_WIDTH	35
5.1.2.13 MAX_GLYPHS	35
5.1.2.14 MAX_NAME_IDS	35
5.1.2.15 PRI_CP	35
5.1.2.16 PW	35
5.1.2.17 static_assert	36
5.1.2.18 U16MAX	36
5.1.2.19 U32MAX	36
5.1.2.20 VERSION	36
5.1.3 Typedef Documentation	36
5.1.3.1 Buffer	37
5.1.3.2 byte	37
5.1.3.3 Glyph	37
5.1.3.4 NameStrings	37
5.1.3.5 Options	37
5.1.3.6 pixels_t	38
5.1.3.7 Table	38
5.1.4 Enumeration Type Documentation	38
5.1.4.1 ContourOp	38
5.1.4.2 FillSide	39
5.1.4.3 LocaFormat	39
5.1.5 Function Documentation	40
5.1.5.1 addTable()	40
5.1.5.2 buildOutline()	42
5.1.5.3 byCodePoint()	45
5.1.5.4 byTableTag()	46
5.1.5.5 cacheBuffer()	47
5.1.5.6 cacheBytes()	47
5.1.5.7 cacheCFFOperand()	49
5.1.5.8 cacheStringAsUTF16BE()	51
5.1.5.9 cacheU16()	52
5.1.5.10 cacheU32()	53
5.1.5.11 cacheU8()	55
5.1.5.12 cacheZeros()	56
5.1.5.13 cleanBuffers()	57
5.1.5.14 defineStore()	58
5.1.5.15 ensureBuffer()	58
5.1.5.16 fail()	60

5.1.5.17 fillBitmap()	62
5.1.5.18 fillBlankOutline()	64
5.1.5.19 fillCFF()	65
5.1.5.20 fillCmapTable()	70
5.1.5.21 fillGposTable()	72
5.1.5.22 fillGsubTable()	73
5.1.5.23 fillHeadTable()	75
5.1.5.24 fillHheaTable()	77
5.1.5.25 fillHmtxTable()	79
5.1.5.26 fillMaxpTable()	80
5.1.5.27 fillNameTable()	82
5.1.5.28 fillOS2Table()	84
5.1.5.29 fillPostTable()	86
5.1.5.30 fillTrueType()	88
5.1.5.31 freeBuffer()	90
5.1.5.32 initBuffers()	91
5.1.5.33 main()	92
5.1.5.34 matchToken()	95
5.1.5.35 newBuffer()	96
5.1.5.36 organizeTables()	98
5.1.5.37 parseOptions()	99
5.1.5.38 positionGlyphs()	101
5.1.5.39 prepareOffsets()	103
5.1.5.40 prepareStringIndex()	104
5.1.5.41 printHelp()	106
5.1.5.42 printVersion()	106
5.1.5.43 readCodePoint()	107
5.1.5.44 readGlyphs()	108
5.1.5.45 sortGlyphs()	110
5.1.5.46 writeBytes()	111
5.1.5.47 writeFont()	113
5.1.5.48 writeU16()	115
5.1.5.49 writeU32()	116
5.1.6 Variable Documentation	117
5.1.6.1 allBuffers	117
5.1.6.2 bufferCount	117
5.1.6.3 nextBufferIndex	117
5.2 hex2otf.c	118
5.3 src/hex2otf.h File Reference	150
5.3.1 Detailed Description	152

5.3.2 Macro Definition Documentation	152
5.3.2.1 DEFAULT_ID0	152
5.3.2.2 DEFAULT_ID1	153
5.3.2.3 DEFAULT_ID11	153
5.3.2.4 DEFAULT_ID13	153
5.3.2.5 DEFAULT_ID14	153
5.3.2.6 DEFAULT_ID2	153
5.3.2.7 DEFAULT_ID5	154
5.3.2.8 NAMEPAIR	154
5.3.2.9 UNIFONT_VERSION	154
5.3.3 Variable Documentation	154
5.3.3.1 defaultNames	154
5.4 hex2otf.h	155
5.5 src/unibdf2hex.c File Reference	156
5.5.1 Detailed Description	157
5.5.2 Macro Definition Documentation	157
5.5.2.1 MAXBUF	157
5.5.2.2 UNISTART	158
5.5.2.3 UNISTOP	158
5.5.3 Function Documentation	158
5.5.3.1 main()	158
5.6 unibdf2hex.c	159
5.7 src/unibmp2hex.c File Reference	161
5.7.1 Detailed Description	162
5.7.2 Macro Definition Documentation	162
5.7.2.1 MAXBUF	162
5.7.3 Function Documentation	163
5.7.3.1 main()	163
5.7.4 Variable Documentation	170
5.7.4.1 bits_per_pixel	170
5.7.4.2	171
5.7.4.3 color_table	171
5.7.4.4 compression	171
5.7.4.5 file_size	171
5.7.4.6 filetype	171
5.7.4.7 flip	172
5.7.4.8 forceline	172
5.7.4.9 height	172
5.7.4.10 hexdigit	172
5.7.4.11 image_offset	172

5.7.4.12	image_size	173
5.7.4.13	important_colors	173
5.7.4.14	info_size	173
5.7.4.15	ncolors	173
5.7.4.16	nplanes	173
5.7.4.17	planeset	173
5.7.4.18	unidigit	174
5.7.4.19	uniplane	174
5.7.4.20	width	174
5.7.4.21	x_ppm	174
5.7.4.22	y_ppm	174
5.8	unibmp2hex.c	175
5.9	src/unibmpbump.c File Reference	184
5.9.1	Detailed Description	184
5.9.2	Macro Definition Documentation	185
5.9.2.1	MAX_COMPRESSION_METHOD	185
5.9.2.2	VERSION	185
5.9.3	Function Documentation	185
5.9.3.1	get_bytes()	185
5.9.3.2	main()	186
5.9.3.3	regrid()	192
5.10	unibmpbump.c	194
5.11	src/unicoverage.c File Reference	201
5.11.1	Detailed Description	202
5.11.2	Macro Definition Documentation	202
5.11.2.1	MAXBUF	202
5.11.3	Function Documentation	202
5.11.3.1	main()	202
5.11.3.2	nextrange()	205
5.11.3.3	print_subtotal()	207
5.12	unicoverage.c	208
5.13	src/unidup.c File Reference	211
5.13.1	Detailed Description	212
5.13.2	Macro Definition Documentation	213
5.13.2.1	MAXBUF	213
5.13.3	Function Documentation	213
5.13.3.1	main()	213
5.14	unidup.c	214
5.15	src/unifont1per.c File Reference	215
5.15.1	Detailed Description	216

5.15.2 Macro Definition Documentation	216
5.15.2.1 MAXFILENAME	216
5.15.2.2 MAXSTRING	216
5.15.3 Function Documentation	217
5.15.3.1 main()	217
5.16 unifont1per.c	218
5.17 src/unifontpic.c File Reference	221
5.17.1 Detailed Description	222
5.17.2 Macro Definition Documentation	222
5.17.2.1 HDR_LEN	222
5.17.3 Function Documentation	222
5.17.3.1 genlongbmp()	222
5.17.3.2 genwidebmp()	227
5.17.3.3 gethex()	233
5.17.3.4 main()	234
5.17.3.5 output2()	236
5.17.3.6 output4()	237
5.18 unifontpic.c	238
5.19 src/unifontpic.h File Reference	249
5.19.1 Detailed Description	250
5.19.2 Macro Definition Documentation	251
5.19.2.1 HEADER_STRING	251
5.19.2.2 MAXSTRING	251
5.19.3 Variable Documentation	251
5.19.3.1 ascii_bits	251
5.19.3.2 ascii_hex	251
5.19.3.3 hexdigit	252
5.20 unifontpic.h	252
5.21 src/unigencircles.c File Reference	255
5.21.1 Detailed Description	256
5.21.2 Macro Definition Documentation	256
5.21.2.1 MAXSTRING	256
5.21.3 Function Documentation	256
5.21.3.1 add_double_circle()	256
5.21.3.2 add_single_circle()	258
5.21.3.3 main()	259
5.22 unigencircles.c	261
5.23 src/unigenwidth.c File Reference	265
5.23.1 Detailed Description	266
5.23.2 Macro Definition Documentation	267

5.23.2.1 MAXSTRING	267
5.23.2.2 PIKTO_END	267
5.23.2.3 PIKTO_SIZE	267
5.23.2.4 PIKTO_START	267
5.23.3 Function Documentation	267
5.23.3.1 main()	267
5.24 unigenwidth.c	272
5.25 src/unihex2bmp.c File Reference	276
5.25.1 Detailed Description	278
5.25.2 Macro Definition Documentation	278
5.25.2.1 MAXBUF	278
5.25.3 Function Documentation	278
5.25.3.1 hex2bit()	278
5.25.3.2 init()	280
5.25.3.3 main()	282
5.25.4 Variable Documentation	286
5.25.4.1 flip	286
5.25.4.2 hex	287
5.25.4.3 hexbits	287
5.25.4.4 unipage	288
5.26 unihex2bmp.c	288
5.27 src/unihexgen.c File Reference	294
5.27.1 Detailed Description	295
5.27.2 Function Documentation	296
5.27.2.1 hexprint4()	296
5.27.2.2 hexprint6()	297
5.27.2.3 main()	299
5.27.3 Variable Documentation	300
5.27.3.1 hexdigit	301
5.28 unihexgen.c	302
5.29 src/unipagecount.c File Reference	305
5.29.1 Detailed Description	306
5.29.2 Macro Definition Documentation	306
5.29.2.1 MAXBUF	307
5.29.3 Function Documentation	307
5.29.3.1 main()	307
5.29.3.2 mkftable()	309
5.30 unipagecount.c	311
Index	315

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 Font \leftrightarrow Forge.

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

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

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

1.4 The C Programs

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

Pro-gram	De-scrip-tion
hex2otf.c	Convert a GNU Unifont .hex file to an Open \leftrightarrow Type font
unibdf2hex.c	Convert a BDF file into a uni-font. \leftrightarrow hex file

Pro-gram	De-scrip-tion
unibmp2t1.c	Turn a .bmp or .wbmp glyph matrix into a GNU Uni-font hex glyph set of 256 characters
unibmp2png.c	Adjust a Microsoft bitmap (.bmp) file that was created by uni-hex2png but converted to .bmp
unicoverage.c	Show the coverage of Uni-code plane scripts for a GNU Uni-font hex glyph file

Pro-gram	De-scrip-tion
unidup.c	Check for duplicate code points in sorted uni-font.↵ hex file
unifont1p.c	Read a Uni-font .hex file from stan-dard input and pro-duce one glyph per .bmp bitmap file as out-put
unifontpic.c	See the "Big Pic-ture"↵ : the entire Uni-font in one BMP bitmap

Pro-gram	De-scrip-tion
unigencircles	Superimpose dashed combining circles on combining glyphs
unigenwidth	IEEE 1003.↵ 1-2008 setup to calculate wchar↵ _t string widths
unihex2b	Turn a GNU Uni-font hex glyph page of 256 code points into a bitmap for editing
unihexgen	Generate a series of glyphs containing hex-adec-imal code points

Pro-gram	De-scrip-tion
unipagecount	Count the number of glyphs defined in each page of 256 code points

1.5 Perl Scripts

The very first program written for Unifont conversion was Roman Czyborra's hexdraw Perl script. That one script would convert a unifont.hex file into a text file with 16 lines per glyph (one for each glyph row) followed by a blank line 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	De-scrip-tion
bdfimplot	Convert a BDF font into GNU Unifont .hex format

Perl Script	Description
hex2bdf	Convert a GNU Uni-font .hex file into a BDF font
hex2sfd	Convert a GNU Uni-font .hex file into a Font \leftrightarrow Forge .sfd format
hexbraille	Algorithmically generate the Uni-code Braille range (U+28xx)
hexdraw	Convert a GNU Uni-font .hex file to and from an ASCII text file

Perl Script	De- scrip- tion
hexkinya	Create the Pri- vate Use Area Kinya sylla- bles
hexmerge	Merge two or more GNU Uni- font .hex font files into one
johab2ucs2	Convert a Jo- hab BDF font into GNU Uni- font Hangul Sylla- bles
unifont- viewer	View a .hex font file with a graph- ical user inter- face

Perl Script	Description
unifontchdir.pl	Extract Hangul syllables that have no final consonant
unifontksx.pl	Extract Hangul syllables that comprise KS X 1001↔:1992
unihex2png.pl	GNU Uni-font .hex file to Portable Network Graphics converter
unihexfill.pl	Generate range of Uni-font 4- or 6-digit hexadecimal glyph

Perl Script	De- scrip- tion
unihexrot.pl	Rotate Uni- font hex glyphs in quar- ter turn incre- ments
unipng2hex.pl	Portable Net- work Graph- ics to GNU Uni- font .hex file con- verter

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	15
Font	Data structure to hold information for one font	16
Glyph	Data structure to hold data for one bitmap glyph	18
NamePair	Data structure for a font ID number and name character string	20
Options	Data structure to hold options for OpenType font output	21
Table	Data structure for an OpenType table	23
TableRecord	Data structure for data associated with one OpenType table	24

Chapter 3

File Index

3.1 File List

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

src/ hex2otf.c	
Hex2otf - Convert GNU Unifont .hex file to OpenType font	27
src/ hex2otf.h	
Hex2otf.h - Header file for hex2otf.c	150
src/ unibdf2hex.c	
Unibdf2hex - Convert a BDF file into a unifont.hex file	156
src/ unibmp2hex.c	
Unibmp2hex - Turn a .bmp or .wbmp glyph matrix into a GNU Unifont hex glyph set of 256 characters	161
src/ unibmpbump.c	
Unibmpbump - Adjust a Microsoft bitmap (.bmp) file that was created by unihex2png but converted to .bmp	184
src/ unicoverage.c	
Unicoverage - Show the coverage of Unicode plane scripts for a GNU Unifont hex glyph file	201
src/ unidup.c	
Unidup - Check for duplicate code points in sorted unifont.hex file	211
src/ unifont1per.c	
Unifont1per - Read a Unifont .hex file from standard input and produce one glyph per ".bmp" bitmap file as output	215
src/ unifontpic.c	
Unifontpic - See the "Big Picture": the entire Unifont in one BMP bitmap	221
src/ unifontpic.h	
Unifontpic.h - Header file for unifontpic.c	249
src/ unigencircles.c	
Unigencircles - Superimpose dashed combining circles on combining glyphs	255
src/ unigenwidth.c	
Unigenwidth - IEEE 1003.1-2008 setup to calculate wchar_t string widths	265
src/ unihex2bmp.c	
Unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points into a bitmap for editing	276
src/ unihexgen.c	
Unihexgen - Generate a series of glyphs containing hexadecimal code points	294
src/ unipagecount.c	
Unipagecount - Count the number of glyphs defined in each page of 256 code points . . .	305

Chapter 4

Data Structure Documentation

4.1 Buffer Struct Reference

Generic data structure for a linked list of buffer elements.

Data Fields

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

4.1.1 Detailed Description

Generic data structure for a linked list of buffer elements.

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

Definition at line 133 of file [hex2otf.c](#).

4.1.2 Field Documentation

4.1.2.1 begin

`byte*` Buffer::begin

Definition at line 136 of file [hex2otf.c](#).

4.1.2.2 capacity

size_t Buffer::capacity

Definition at line 135 of file [hex2otf.c](#).

4.1.2.3 end

byte * Buffer::end

Definition at line 136 of file [hex2otf.c](#).

4.1.2.4 next

byte * Buffer::next

Definition at line 136 of file [hex2otf.c](#).

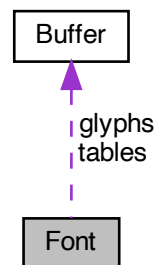
The documentation for this struct was generated from the following file:

- [src/hex2otf.c](#)

4.2 Font Struct Reference

Data structure to hold information for one font.

Collaboration diagram for Font:



Data Fields

- [Buffer * tables](#)
- [Buffer * glyphs](#)
- [uint_fast32_t glyphCount](#)
- [pixels_t maxWidth](#)

4.2.1 Detailed Description

Data structure to hold information for one font.

Definition at line [628](#) of file [hex2otf.c](#).

4.2.2 Field Documentation

4.2.2.1 glyphCount

[uint_fast32_t](#) Font::glyphCount

Definition at line [632](#) of file [hex2otf.c](#).

4.2.2.2 glyphs

[Buffer*](#) Font::glyphs

Definition at line [631](#) of file [hex2otf.c](#).

4.2.2.3 maxWidth

[pixels_t](#) Font::maxWidth

Definition at line [633](#) of file [hex2otf.c](#).

4.2.2.4 tables

[Buffer*](#) Font::tables

Definition at line [630](#) of file [hex2otf.c](#).

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

- [src/hex2otf.c](#)

4.3 Glyph Struct Reference

Data structure to hold data for one bitmap glyph.

Data Fields

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

4.3.1 Detailed Description

Data structure to hold data for one bitmap glyph.

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

Definition at line [614](#) of file [hex2otf.c](#).

4.3.2 Field Documentation

4.3.2.1 bitmap

`byte` Glyph::bitmap[GLYPH_MAX_BYTE_COUNT]

hexadecimal bitmap character array

Definition at line 617 of file [hex2otf.c](#).

4.3.2.2 byteCount

`uint_least8_t` Glyph::byteCount

length of bitmap data

Definition at line 618 of file [hex2otf.c](#).

4.3.2.3 codePoint

`uint_least32_t` Glyph::codePoint

undefined for glyph 0

Definition at line 616 of file [hex2otf.c](#).

4.3.2.4 combining

`bool` Glyph::combining

whether this is a combining glyph

Definition at line 619 of file [hex2otf.c](#).

4.3.2.5 lsb

`pixels_t` Glyph::lsb

left side bearing (x position of leftmost contour point)

Definition at line 622 of file [hex2otf.c](#).

4.3.2.6 pos

[pixels_t](#) Glyph::pos

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

Definition at line [620](#) of file [hex2otf.c](#).

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

- [src/hex2otf.c](#)

4.4 NamePair Struct Reference

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

```
#include <hex2otf.h>
```

Data Fields

- int [id](#)
- const char * [str](#)

4.4.1 Detailed Description

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

Definition at line [77](#) of file [hex2otf.h](#).

4.4.2 Field Documentation

4.4.2.1 id

int NamePair::id

Definition at line [79](#) of file [hex2otf.h](#).

4.4.2.2 str

const char* NamePair::str

Definition at line 80 of file [hex2otf.h](#).

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

- [src/hex2otf.h](#)

4.5 Options Struct Reference

Data structure to hold options for OpenType font output.

Data Fields

- bool [truetype](#)
- bool [blankOutline](#)
- bool [bitmap](#)
- bool [gpos](#)
- bool [gsub](#)
- int [cff](#)
- const char * [hex](#)
- const char * [pos](#)
- const char * [out](#)
- [NameStrings](#) [nameStrings](#)

4.5.1 Detailed Description

Data structure to hold options for OpenType font output.

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

Definition at line 2453 of file [hex2otf.c](#).

4.5.2 Field Documentation

4.5.2.1 bitmap

bool Options::bitmap

Definition at line 2455 of file [hex2otf.c](#).

4.5.2.2 blankOutline

bool Options::blankOutline

Definition at line [2455](#) of file [hex2otf.c](#).

4.5.2.3 cff

int Options::cff

Definition at line [2456](#) of file [hex2otf.c](#).

4.5.2.4 gpos

bool Options::gpos

Definition at line [2455](#) of file [hex2otf.c](#).

4.5.2.5 gsub

bool Options::gsub

Definition at line [2455](#) of file [hex2otf.c](#).

4.5.2.6 hex

const char* Options::hex

Definition at line [2457](#) of file [hex2otf.c](#).

4.5.2.7 nameStrings

[NameStrings](#) Options::nameStrings

Definition at line [2458](#) of file [hex2otf.c](#).

4.5.2.8 out

```
const char * Options::out
```

Definition at line [2457](#) of file [hex2otf.c](#).

4.5.2.9 pos

```
const char * Options::pos
```

Definition at line [2457](#) of file [hex2otf.c](#).

4.5.2.10 truetype

```
bool Options::truetype
```

Definition at line [2455](#) of file [hex2otf.c](#).

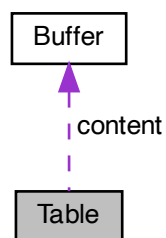
The documentation for this struct was generated from the following file:

- [src/hex2otf.c](#)

4.6 Table Struct Reference

Data structure for an OpenType table.

Collaboration diagram for Table:



Data Fields

- `uint_fast32_t tag`
- `Buffer * content`

4.6.1 Detailed Description

Data structure for an OpenType table.

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

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

Definition at line 645 of file `hex2otf.c`.

4.6.2 Field Documentation

4.6.2.1 content

`Buffer* Table::content`

Definition at line 648 of file `hex2otf.c`.

4.6.2.2 tag

`uint_fast32_t Table::tag`

Definition at line 647 of file `hex2otf.c`.

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

- `src/hex2otf.c`

4.7 TableRecord Struct Reference

Data structure for data associated with one OpenType table.

Data Fields

- `uint_least32_t` [tag](#)
- `uint_least32_t` [offset](#)
- `uint_least32_t` [length](#)
- `uint_least32_t` [checksum](#)

4.7.1 Detailed Description

Data structure for data associated with one OpenType table.

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

Definition at line [747](#) of file [hex2otf.c](#).

4.7.2 Field Documentation

4.7.2.1 checksum

`uint_least32_t` `TableRecord::checksum`

Definition at line [749](#) of file [hex2otf.c](#).

4.7.2.2 length

`uint_least32_t` `TableRecord::length`

Definition at line [749](#) of file [hex2otf.c](#).

4.7.2.3 offset

`uint_least32_t` `TableRecord::offset`

Definition at line [749](#) of file [hex2otf.c](#).

4.7.2.4 tag

`uint_least32_t` `TableRecord::tag`

Definition at line [749](#) of file [hex2otf.c](#).

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

- `src/hex2otf.c`

Chapter 5

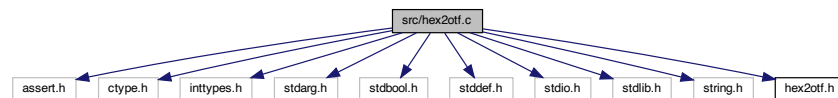
File Documentation

5.1 src/hex2otf.c File Reference

hex2otf - Convert GNU Unifont .hex file to OpenType font

```
#include <assert.h>
#include <ctype.h>
#include <inttypes.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hex2otf.h"
```

Include dependency graph for hex2otf.c:



Data Structures

- struct [Buffer](#)
Generic data structure for a linked list of buffer elements.
- struct [Glyph](#)
Data structure to hold data for one bitmap glyph.
- struct [Font](#)
Data structure to hold information for one font.
- struct [Table](#)

- Data structure for an OpenType table.
- struct [TableRecord](#)
Data structure for data associated with one OpenType table.
- struct [Options](#)
Data structure to hold options for OpenType font output.

Macros

- #define [VERSION](#) "1.0.1"
Program version, for "--version" option.
- #define [U16MAX](#) 0xffff
Maximum UTF-16 code point value.
- #define [U32MAX](#) 0xffffffff
Maximum UTF-32 code point value.
- #define [PRI_CP](#) "U+%.4"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 [FUPPM](#) 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) * [FUPPM](#) / [GLYPH_HEIGHT](#))
Convert pixels to font units.
- #define [PW](#)(x) ((x) / ([GLYPH_HEIGHT](#) / 8))
Convert glyph byte count to pixel width.
- #define [defineStore](#)(name, type)
Temporary define to look up an element in an array of given type.
- #define [addByte](#)(shift)
- #define [getRowBit](#)(rows, x, y) ((rows)[(y)] & x0 >> (x))
- #define [flipRowBit](#)(rows, x, y) ((rows)[(y)] ^ x0 >> (x))
- #define [stringCount](#)(sizeof strings / sizeof *strings)
- #define [cacheCFF32](#)(buf, x) ([cacheU8](#) ((buf), 29), [cacheU32](#) ((buf), (x)))

Typedefs

- typedef unsigned char [byte](#)
Definition of "byte" type as an unsigned char.
- typedef int_least8_t [pixels_t](#)
This type must be able to represent max(GLYPH_MAX_WIDTH, GLYPH_HEIGHT).
- typedef struct [Buffer](#) [Buffer](#)
Generic data structure for a linked list of buffer elements.
- typedef const char * [NameStrings](#)[MAX_NAME_IDS]
Array of OpenType names indexed directly by Name IDs.
- typedef struct [Glyph](#) [Glyph](#)
Data structure to hold data for one bitmap glyph.
- typedef struct [Font](#) [Font](#)
Data structure to hold information for one font.
- typedef struct [Table](#) [Table](#)
Data structure for an OpenType table.
- typedef struct [Options](#) [Options](#)
Data structure to hold options for OpenType font output.

Enumerations

- enum [LocaFormat](#) { [LOCA_OFFSET16](#) = 0 , [LOCA_OFFSET32](#) = 1 }
Index to Location ("loca") offset information.
- enum [ContourOp](#) { [OP_CLOSE](#) , [OP_POINT](#) }
Specify the current contour drawing operation.
- enum [FillSide](#) { [FILL_LEFT](#) , [FILL_RIGHT](#) }
Fill to the left side (CFF) or right side (TrueType) of a contour.

Functions

- void [fail](#) (const char *reason,...)
Print an error message on stderr, then exit.
- void [initBuffers](#) (size_t count)
Initialize an array of buffer pointers to all zeroes.
- void [cleanBuffers](#) ()
Free all allocated buffer pointers.
- [Buffer](#) * [newBuffer](#) (size_t initialCapacity)
Create a new buffer.
- void [ensureBuffer](#) ([Buffer](#) *buf, size_t needed)
Ensure that the buffer has at least the specified minimum size.
- void [freeBuffer](#) ([Buffer](#) *buf)
Free the memory previously allocated for a buffer.
- [defineStore](#) (storeU8, uint_least8_t)
- void [cacheU8](#) ([Buffer](#) *buf, uint_fast8_t value)
Append one unsigned byte to the end of a byte array.
- void [cacheU16](#) ([Buffer](#) *buf, uint_fast16_t value)

- Append two unsigned bytes to the end of a byte array.
- void `cacheU32` (`Buffer` *buf, `uint_fast32_t` value)
- Append four unsigned bytes to the end of a byte array.
- void `cacheCFFOperand` (`Buffer` *buf, `int_fast32_t` value)
- Cache charstring number encoding in a CFF buffer.
- void `cacheZeros` (`Buffer` *buf, `size_t` count)
- Append 1 to 4 bytes of zeroes to a buffer, for padding.
- void `cacheBytes` (`Buffer` *restrict buf, const void *restrict src, `size_t` count)
- Append a string of bytes to a buffer.
- void `cacheBuffer` (`Buffer` *restrict bufDest, const `Buffer` *restrict bufSrc)
- Append bytes of a table to a byte buffer.
- void `writeBytes` (const `byte` bytes[], `size_t` count, `FILE` *file)
- Write an array of bytes to an output file.
- void `writeU16` (`uint_fast16_t` value, `FILE` *file)
- Write an unsigned 16-bit value to an output file.
- void `writeU32` (`uint_fast32_t` value, `FILE` *file)
- Write an unsigned 32-bit value to an output file.
- void `addTable` (`Font` *font, const char tag[static 4], `Buffer` *content)
- Add a TrueType or OpenType table to the font.
- void `organizeTables` (`Font` *font, bool isCFF)
- Sort tables according to OpenType recommendations.
- int `byTableTag` (const void *a, const void *b)
- Compare tables by 4-byte unsigned table tag value.
- void `writeFont` (`Font` *font, bool isCFF, const char *fileName)
- Write OpenType font to output file.
- bool `readCodePoint` (`uint_fast32_t` *codePoint, const char *fileName, `FILE` *file)
- Read up to 6 hexadecimal digits and a colon from file.
- void `readGlyphs` (`Font` *font, const char *fileName)
- Read glyph definitions from a Unifont .hex format file.
- int `byCodePoint` (const void *a, const void *b)
- Compare two Unicode code points to determine which is greater.
- void `positionGlyphs` (`Font` *font, const char *fileName, `pixels_t` *xMin)
- Position a glyph within a 16-by-16 pixel bounding box.
- void `sortGlyphs` (`Font` *font)
- Sort the glyphs in a font by Unicode code point.
- void `buildOutline` (`Buffer` *result, const `byte` bitmap[], const `size_t` byteCount, const enum `FillSide` fillSide)
- Build a glyph outline.
- void `prepareOffsets` (`size_t` *sizes)
- Prepare 32-bit glyph offsets in a font table.
- `Buffer` * `prepareStringIndex` (const `NameStrings` names)
- Prepare a font name string index.
- void `fillCFF` (`Font` *font, int version, const `NameStrings` names)
- Add a CFF table to a font.
- void `fillTrueType` (`Font` *font, enum `LocaFormat` *format, `uint_fast16_t` *maxPoints, `uint_fast16_t` *maxContours)
- Add a TrueType table to a font.

- void [fillBlankOutline](#) ([Font](#) *font)

Create a dummy blank outline in a font table.
- void [fillBitmap](#) ([Font](#) *font)

Fill OpenType bitmap data and location tables.
- void [fillHeadTable](#) ([Font](#) *font, enum [LocaFormat](#) locaFormat, [pixels_t](#) xMin)

Fill a "head" font table.
- void [fillHheaTable](#) ([Font](#) *font, [pixels_t](#) xMin)

Fill a "hhea" font table.
- void [fillMaxpTable](#) ([Font](#) *font, bool isCFF, [uint_fast16_t](#) maxPoints, [uint_fast16_t](#) maxContours)

Fill a "maxp" font table.
- void [fillOS2Table](#) ([Font](#) *font)

Fill an "OS/2" font table.
- void [fillHmtxTable](#) ([Font](#) *font)

Fill an "hmtx" font table.
- void [fillCmapTable](#) ([Font](#) *font)

Fill a "cmap" font table.
- void [fillPostTable](#) ([Font](#) *font)

Fill a "post" font table.
- void [fillGposTable](#) ([Font](#) *font)

Fill a "GPOS" font table.
- void [fillGsubTable](#) ([Font](#) *font)

Fill a "GSUB" font table.
- void [cacheStringAsUTF16BE](#) ([Buffer](#) *buf, const char *str)

Cache a string as a big-ending UTF-16 surrogate pair.
- void [fillNameTable](#) ([Font](#) *font, [NameStrings](#) nameStrings)

Fill a "name" font table.
- void [printVersion](#) ()

Print program version string on stdout.
- void [printHelp](#) ()

Print help message to stdout and then exit.
- const char * [matchToken](#) (const char *operand, const char *key, char delimiter)

Match a command line option with its key for enabling.
- [Options](#) [parseOptions](#) (char *const argv[const])

Parse command line options.
- int [main](#) (int argc, char *argv[])

The main function.

Variables

- [Buffer](#) * [allBuffers](#)

Initial allocation of empty array of buffer pointers.
- [size_t](#) [bufferCount](#)

Number of buffers in a [Buffer](#) * array.
- [size_t](#) [nextBufferIndex](#)

Index number to tail element of [Buffer](#) * array.

5.1.1 Detailed Description

hex2otf - Convert GNU Unifont .hex file to OpenType font

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

Copyright

Copyright © 2022 何志翔 (He Zhixiang)

Author

何志翔 (He Zhixiang)

Definition in file [hex2otf.c](#).

5.1.2 Macro Definition Documentation

5.1.2.1 addByte

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

Value:

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

5.1.2.2 ASCENDER

```
#define ASCENDER (GLYPH_HEIGHT - DESCENDER)
```

Count of pixels above baseline.

Definition at line [79](#) of file [hex2otf.c](#).

5.1.2.3 B0

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

Clear a given bit in a word.

Definition at line 66 of file [hex2otf.c](#).

5.1.2.4 B1

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

Set a given bit in a word.

Definition at line 67 of file [hex2otf.c](#).

5.1.2.5 BX

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

Truncate & shift word.

Definition at line 65 of file [hex2otf.c](#).

5.1.2.6 defineStore

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

Value:

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

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

This 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](#).

5.1.2.7 DESCENDER

```
#define DESCENDER 2
```

Count of pixels below baseline.

Definition at line 76 of file [hex2otf.c](#).

5.1.2.8 FU

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

Convert pixels to font units.

Definition at line 91 of file [hex2otf.c](#).

5.1.2.9 FUPEM

```
#define FUPEM 64
```

[Font](#) units per em.

Definition at line 82 of file [hex2otf.c](#).

5.1.2.10 GLYPH_HEIGHT

```
#define GLYPH_HEIGHT 16
```

Maximum glyph height, in pixels.

Definition at line 70 of file [hex2otf.c](#).

5.1.2.11 GLYPH_MAX_BYTE_COUNT

```
#define GLYPH_MAX_BYTE_COUNT (GLYPH_HEIGHT * GLYPH_MAX_WIDTH / 8)
```

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

Definition at line 73 of file [hex2otf.c](#).

5.1.2.12 GLYPH_MAX_WIDTH

```
#define GLYPH_MAX_WIDTH 16
```

Maximum glyph width, in pixels.

Definition at line 69 of file [hex2otf.c](#).

5.1.2.13 MAX_GLYPHS

```
#define MAX_GLYPHS 65536
```

An OpenType font has at most 65536 glyphs.

Definition at line 85 of file [hex2otf.c](#).

5.1.2.14 MAX_NAME_IDS

```
#define MAX_NAME_IDS 256
```

Name IDs 0-255 are used for standard names.

Definition at line 88 of file [hex2otf.c](#).

5.1.2.15 PRI_CP

```
#define PRI_CP "U+%.4"PRIxF32
```

Format string to print Unicode code point.

Definition at line 58 of file [hex2otf.c](#).

5.1.2.16 PW

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

Convert glyph byte count to pixel width.

Definition at line 94 of file [hex2otf.c](#).

5.1.2.17 static_assert

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

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

Definition at line [61](#) of file [hex2otf.c](#).

5.1.2.18 U16MAX

```
#define U16MAX 0xffff
```

Maximum UTF-16 code point value.

Definition at line [55](#) of file [hex2otf.c](#).

5.1.2.19 U32MAX

```
#define U32MAX 0xffffffff
```

Maximum UTF-32 code point value.

Definition at line [56](#) of file [hex2otf.c](#).

5.1.2.20 VERSION

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

Program version, for "--version" option.

Definition at line [51](#) of file [hex2otf.c](#).

5.1.3 Typedef Documentation

5.1.3.1 Buffer

typedef struct [Buffer](#) [Buffer](#)

Generic data structure for a linked list of buffer elements.

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

5.1.3.2 byte

typedef unsigned char [byte](#)

Definition of "byte" type as an unsigned char.

Definition at line [97](#) of file [hex2otf.c](#).

5.1.3.3 Glyph

typedef struct [Glyph](#) [Glyph](#)

Data structure to hold data for one bitmap glyph.

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

5.1.3.4 NameStrings

typedef const char* NameStrings[[MAX_NAME_IDS](#)]

Array of OpenType names indexed directly by Name IDs.

Definition at line [604](#) of file [hex2otf.c](#).

5.1.3.5 Options

typedef struct [Options](#) [Options](#)

Data structure to hold options for OpenType font output.

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

5.1.3.6 pixels_t

```
typedef int_least8_t pixels_t
```

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

Definition at line 100 of file [hex2otf.c](#).

5.1.3.7 Table

```
typedef struct Table Table
```

Data structure for an OpenType table.

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

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

5.1.4 Enumeration Type Documentation

5.1.4.1 ContourOp

```
enum ContourOp
```

Specify the current contour drawing operation.

Enumerator

OP_CLOSE	Close the current contour path that was being drawn.
OP_POINT	Add one more (x,y) point to the contour 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 contour being drawn.
01139 };
```

5.1.4.2 FillSide

enum [FillSide](#)

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

Enumerator

FILL_LEFT	Draw out- line counter- clockwise (CFF, PostScript).
FILL_RIGHT	Draw out- line clock- wise (TrueType).

Definition at line 1144 of file [hex2otf.c](#).

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

5.1.4.3 LocaFormat

enum [LocaFormat](#)

Index to Location ("loca") offset information.

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

Enumerator

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

Definition at line 658 of file [hex2otf.c](#).

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

5.1.5 Function Documentation

5.1.5.1 addTable()

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

Add a TrueType or OpenType table to the font.

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

Parameters

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

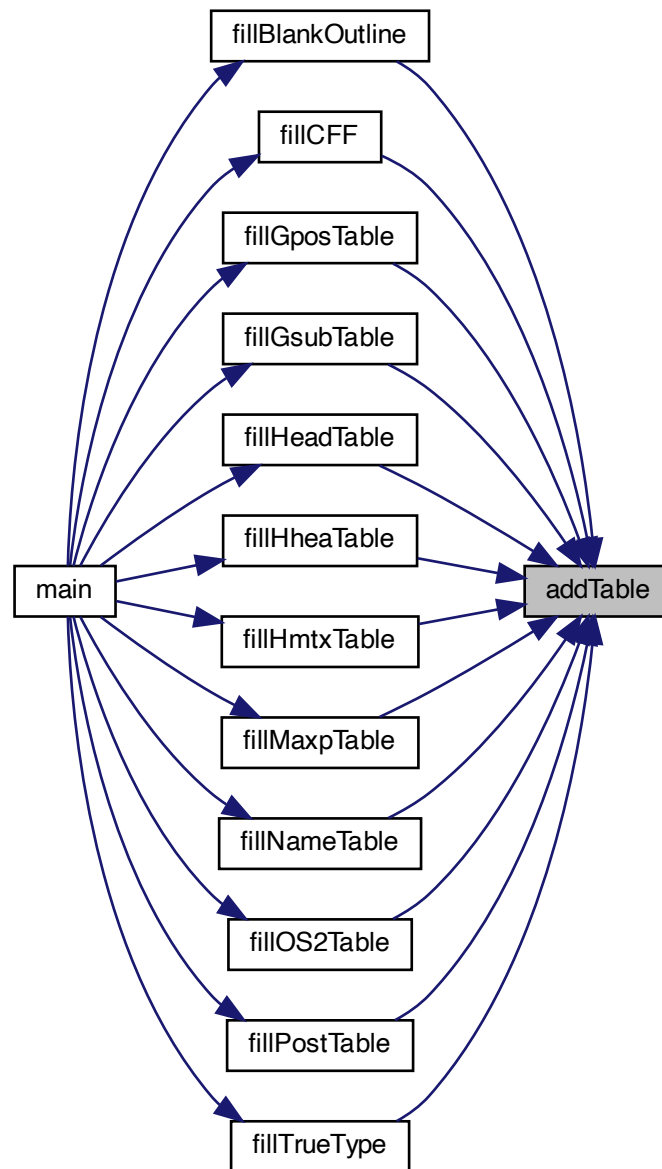
Parameters

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

Definition at line 694 of file [hex2otf.c](#).

```
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:



5.1.5.2 buildOutline()

```
void buildOutline (  
    Buffer * result,
```

```

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

```

Build a glyph outline.

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

Parameters

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

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

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

Definition at line 1160 of file [hex2otf.c](#).

```

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

```

```

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 }

```

5.1.5.3 byCodePoint()

```

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

```

Compare two Unicode code points to determine which is greater.

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

Parameters

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

Returns

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

Definition at line [1040](#) of file [hex2otf.c](#).

```

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 }

```

5.1.5.4 byTableTag()

```

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

```

Compare tables by 4-byte unsigned table tag value.

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

Parameters

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

Returns

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

Definition at line [767](#) of file [hex2otf.c](#).

```

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 }

```


5.1.5.5 cacheBuffer()

```
void cacheBuffer (  
    Buffer *restrict bufDest,  
    const Buffer *restrict bufSrc )
```

Append bytes of a table to a byte buffer.

Parameters

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

Definition at line 523 of file [hex2otf.c](#).

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

5.1.5.6 cacheBytes()

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

Append a string of bytes to a buffer.

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

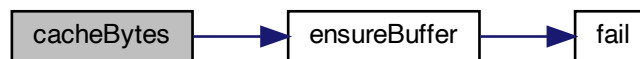
Parameters

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

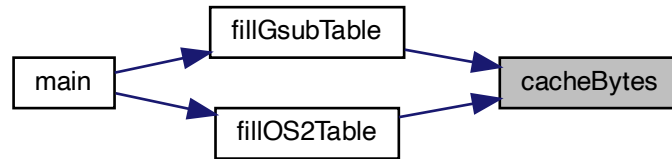
Definition at line 509 of file [hex2otf.c](#).

```
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:



5.1.5.7 cacheCFFOperand()

```

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

Cache charstring number encoding in a CFF buffer.

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

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

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

Parameters

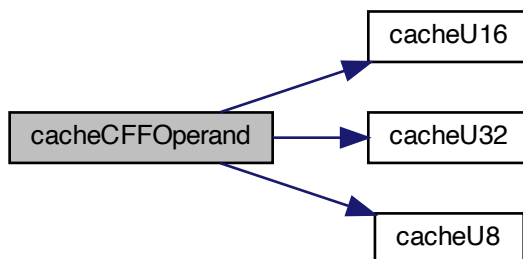
in,out	buf	The buffer to which the operand value is appended.
in	value	The operand value.

Definition at line 460 of file [hex2otf.c](#).

```

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:



5.1.5.8 cacheStringAsUTF16BE()

```
void cacheStringAsUTF16BE (
    Buffer * buf,
    const char * str )
```

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

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

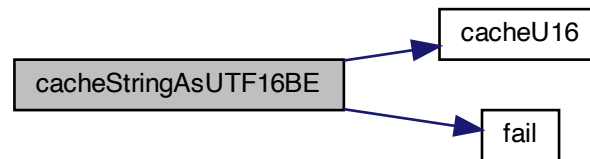
Parameters

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

Definition at line 2316 of file [hex2otf.c](#).

```
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 & 0x3fff));
02351         }
02352         else
02353             cacheU16 (buf, codePoint);
02354     }
02355 }
```

Here is the call graph for this function:



5.1.5.9 cacheU16()

```
void cacheU16 (
    Buffer * buf,
    uint_fast16_t value )
```

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

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

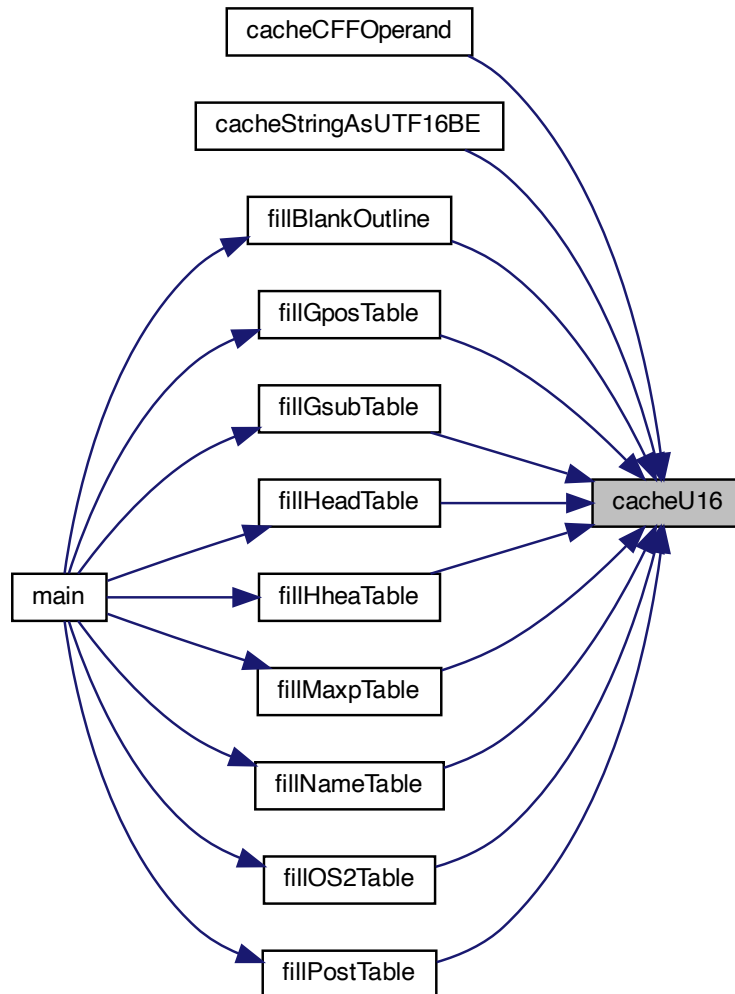
Parameters

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

Definition at line 412 of file [hex2otf.c](#).

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

Here is the caller graph for this function:



5.1.5.10 cacheU32()

```
void cacheU32 (
    Buffer * buf,
    uint_fast32_t value )
```

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

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

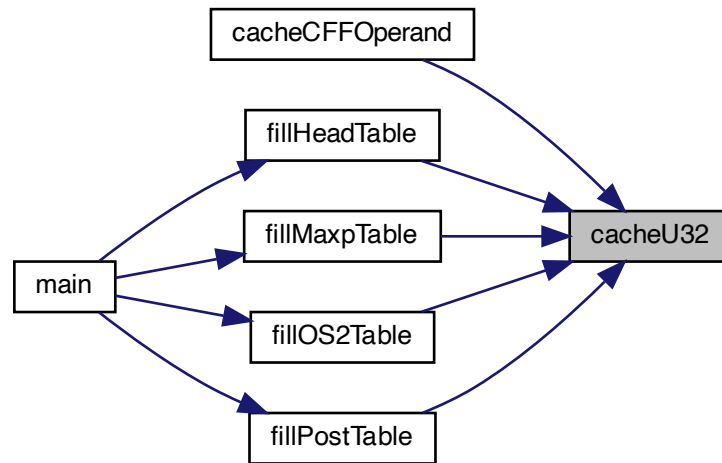
Parameters

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

Definition at line [427](#) of file [hex2otf.c](#).

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


Here is the caller graph for this function:



5.1.5.11 cacheU8()

```
void cacheU8 (
    Buffer * buf,
    uint_fast8_t value )
```

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

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

Parameters

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

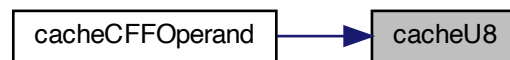
Parameters

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

Definition at line 397 of file [hex2otf.c](#).

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

Here is the caller graph for this function:



5.1.5.12 cacheZeros()

```
void cacheZeros (
    Buffer * buf,
    size_t count )
```

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

Parameters

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

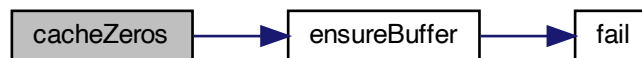
Parameters

in	count	The number of bytes containing zeroes to append.

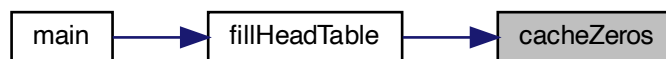
Definition at line 491 of file [hex2otf.c](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.5.13 `cleanBuffers()`

```
void cleanBuffers ( )
```

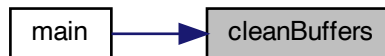
Free all allocated buffer pointers.

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

Definition at line 170 of file `hex2otf.c`.

```
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:



5.1.5.14 `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 }
```

5.1.5.15 `ensureBuffer()`

```
void ensureBuffer (
    Buffer * buf,
    size_t needed )
```

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

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

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

Parameters

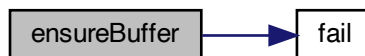
in,out	buf	The buffer to check.
in	needed	The required minimum number of elements in the buffer.

Definition at line 239 of file [hex2otf.c](#).

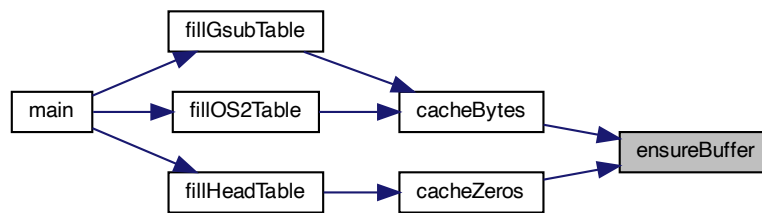
```

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:



5.1.5.16 fail()

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

Print an error message on stderr, then exit.

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

Parameters

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

Definition at line 113 of file [hex2otf.c](#).

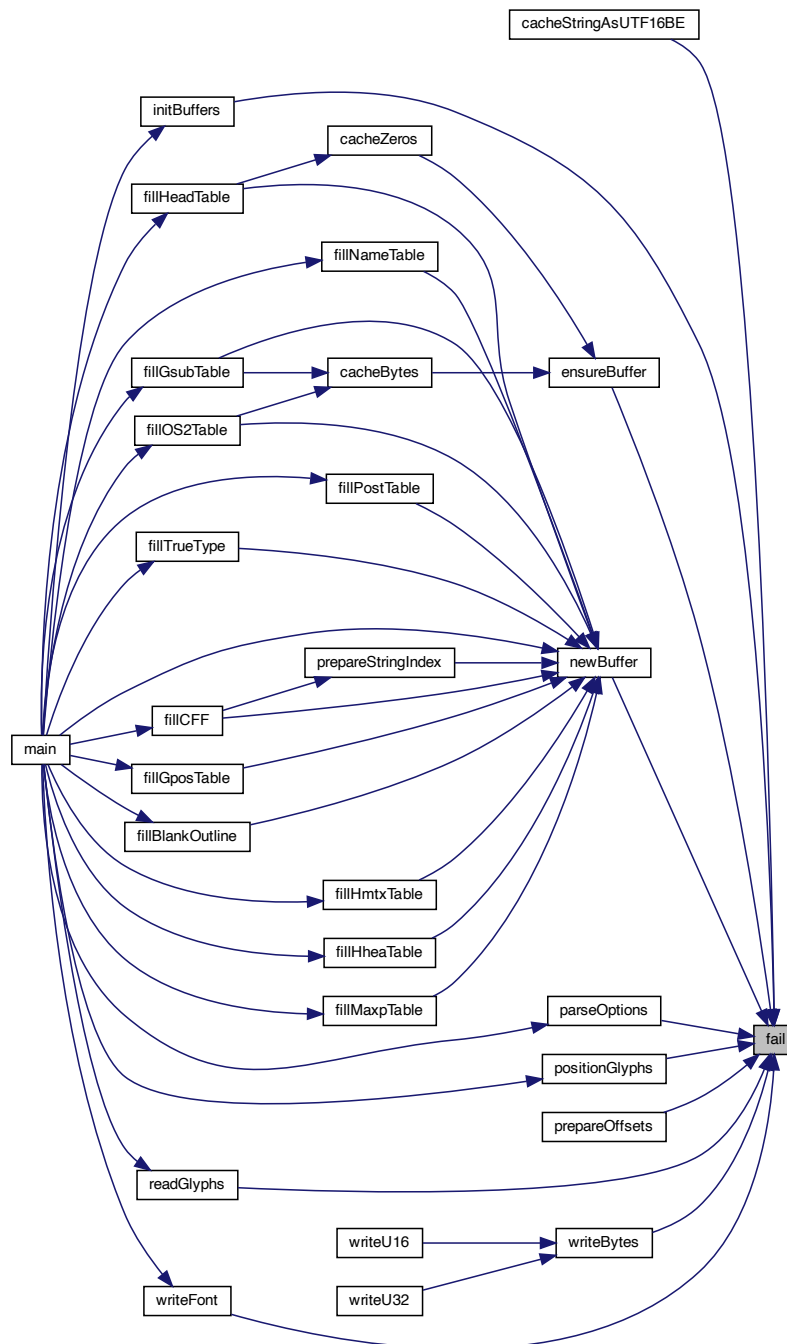
```
00114 {
00115     fputs ("ERROR: ", stderr);
```

```

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

```

Here is the caller graph for this function:



5.1.5.17 fillBitmap()

```
void fillBitmap (
    Font * font )
```

Fill OpenType bitmap data and location tables.

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

Parameters

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

Definition at line 1728 of file [hex2otf.c](#).

```
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 *ebcl = newBuffer (4096);
01762     addTable (font, "EBLC", ebcl);
01763     cacheU16 (ebcl, 2); // majorVersion
01764     cacheU16 (ebcl, 0); // minorVersion
01765     cacheU32 (ebcl, 1); // numSizes
01766     { // bitmapSizes[0]
01767         cacheU32 (ebcl, 56); // indexSubTableArrayOffset
01768         cacheU32 (ebcl, (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 caller graph for this function:



5.1.5.18 fillBlankOutline()

```
void fillBlankOutline (
    Font * font )
```

Create a dummy blank outline in a font table.

Parameters

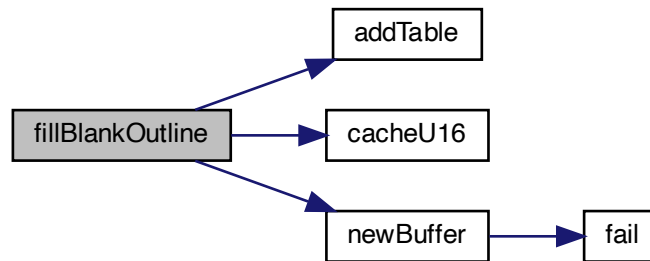
in,out	font	Pointer to a Font struct to insert a blank outline.
--------	------	---

Definition at line 1697 of file [hex2otf.c](#).

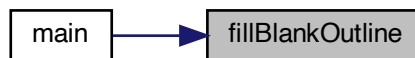
```

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 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.5.19 fillCFF()

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

Add a CFF table to a font.

Parameters

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

Use fixed width integer for variables to simplify offset calculation.

Definition at line 1329 of file [hex2otf.c](#).

```

1330 {
1331     // HACK: For convenience, CFF data structures are hard coded.
1332     assert (0 < version && version <= 2);
1333     Buffer *cff = newBuffer (65536);
1334     addTable (font, version == 1 ? "CFF " : "CFF2", cff);
1335
1336     /// Use fixed width integer for variables to simplify offset calculation.
1337     #define cacheCFF32(buf, x) (cacheU8 ((buf), 29), cacheU32 ((buf), (x)))
1338
1339     // In Unifont, 16px glyphs are more common. This is used by CFF1 only.
1340     const pixels_t defaultWidth = 16, nominalWidth = 8;
1341     if (version == 1)
1342     {
1343         Buffer *strings = prepareStringIndex (names);
1344         size_t stringsSize = countBufferedBytes (strings);
1345         const char *cffName = names[6];
1346         assert (cffName);
1347         size_t nameLength = strlen (cffName);
1348         size_t namesSize = nameLength + 5;
1349         // These sizes must be updated together with the data below.
1350         size_t offsets[] = {4, namesSize, 45, stringsSize, 2, 5, 8, 32, 4, 0};
1351         prepareOffsets (offsets);
1352         { // Header
1353             cacheU8 (cff, 1); // major
1354             cacheU8 (cff, 0); // minor
1355             cacheU8 (cff, 4); // hdrSize
1356             cacheU8 (cff, 1); // offSize
1357         }
1358         assert (countBufferedBytes (cff) == offsets[0]);
1359         { // Name INDEX (should not be used by OpenType readers)
1360             cacheU16 (cff, 1); // count
1361             cacheU8 (cff, 1); // offSize
1362             cacheU8 (cff, 1); // offset[0]
1363             if (nameLength + 1 > 255) // must be too long; spec limit is 63
1364                 fail ("PostScript name is too long.");
1365             cacheU8 (cff, nameLength + 1); // offset[1]
1366             cacheBytes (cff, cffName, nameLength);
1367         }
1368         assert (countBufferedBytes (cff) == offsets[1]);
1369         { // Top DICT INDEX
1370             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             }
01533             assert (!(isDrawing && s == 3));

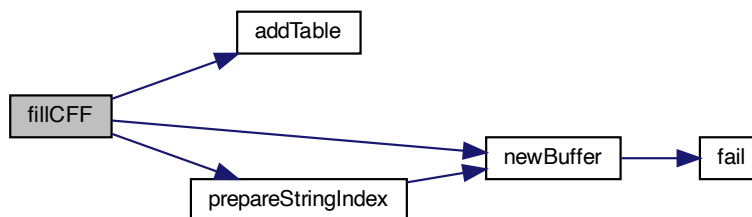
```

```

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); // offsetSize
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:



5.1.5.20 fillCmapTable()

```
void fillCmapTable (
    Font * font )
```

Fill a "cmap" font table.

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

Parameters

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

Definition at line [2109](#) of file [hex2otf.c](#).

```

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

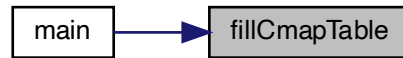


```

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 }

```

Here is the caller graph for this function:



5.1.5.21 fillGposTable()

```
void fillGposTable (
    Font * font )
```

Fill a "GPOS" font table.

The "GPOS" table contains information for glyph positioning.

Parameters

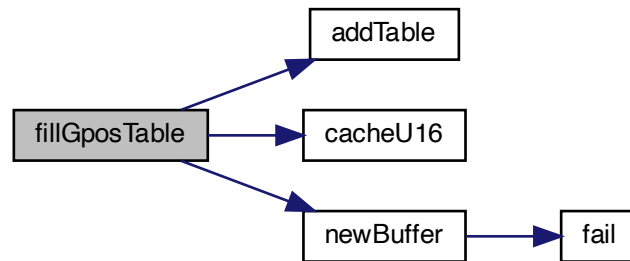
in,out	font	The Font struct to which to add the table.
--------	------	--

Definition at line [2241](#) of file [hex2otf.c](#).

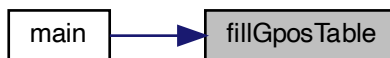
```

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:



5.1.5.22 fillGsubTable()

```
void fillGsubTable (  
    Font * font )
```

Fill a "GSUB" font table.

The "GSUB" table contains information for glyph substitution.

Parameters

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

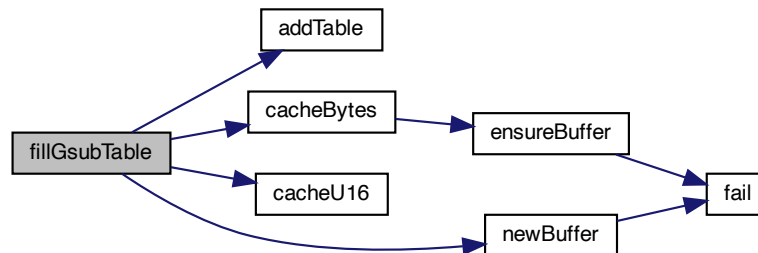
Definition at line 2269 of file [hex2otf.c](#).

```

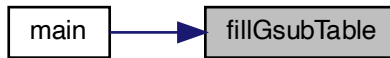
02270 {
02271     Buffer *gsub = newBuffer (38);
02272     addTable (font, "GSUB", gsub);
02273     cacheU16 (gsub, 1); // majorVersion
02274     cacheU16 (gsub, 0); // minorVersion
02275     cacheU16 (gsub, 10); // scriptListOffset
02276     cacheU16 (gsub, 34); // featureListOffset
02277     cacheU16 (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:



5.1.5.23 fillHeadTable()

```

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

Fill a "head" font table.

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

Parameters

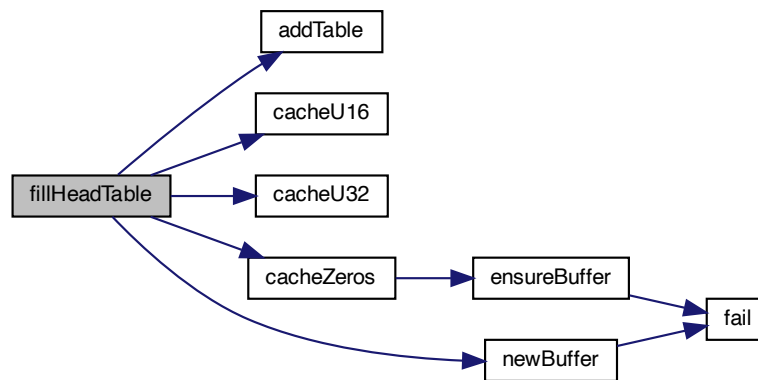
in,out	font	The Font struct to which to add the table.
in	locaFormat	The "loca" offset index location table.
in	xMin	The minimum x-coordinate for a glyph.

Definition at line 1853 of file `hex2otf.c`.

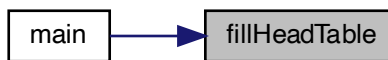
```

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:



5.1.5.24 fillHheaTable()

```
void fillHheaTable (
    Font * font,
    pixels_t xMin )
```

Fill a "hhea" font table.

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

Parameters

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

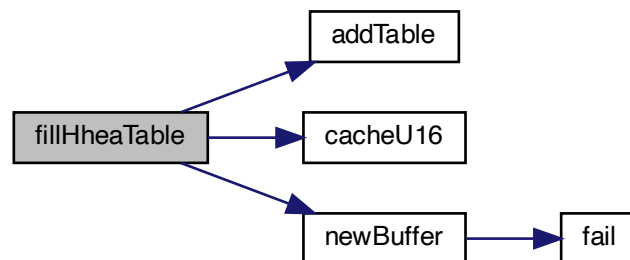
Definition at line 1918 of file [hex2otf.c](#).

```

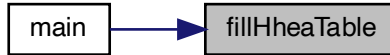
01919 {
01920     Buffer *hhea = newBuffer (36);
01921     addTable (font, "hhea", hhea);
01922     cacheU16 (hhea, 1); // majorVersion
01923     cacheU16 (hhea, 0); // minorVersion
01924     cacheU16 (hhea, FU (ASCENDER)); // ascender
01925     cacheU16 (hhea, FU (-DESCENDER)); // descender
01926  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

```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.5.25 fillHmtxTable()

```
void fillHmtxTable (
    Font * font )
```

Fill an "hmtx" font table.

The "hmtx" table contains horizontal metrics information.

Parameters

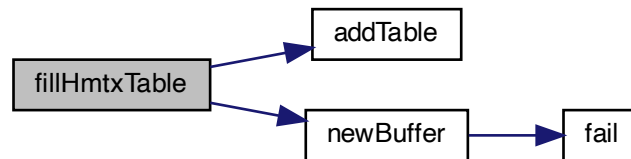
in,out	font	The Font struct to which to add the table.
--------	------	--

Definition at line [2087](#) of file [hex2otf.c](#).

```

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:



5.1.5.26 fillMaxpTable()

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

Fill a "maxp" font table.

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

Parameters

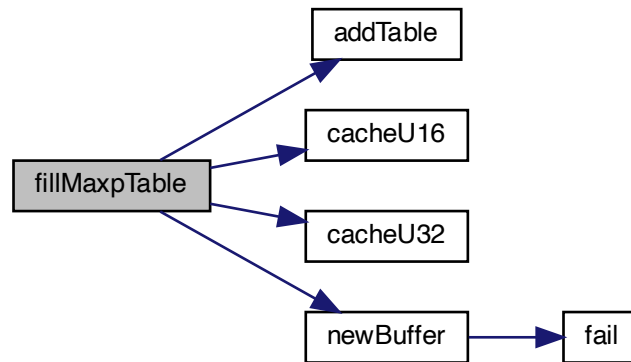
in,out	font	The Font struct to which to add the table.
in	isCFF	true if a CFF font is included, false otherwise.
in	maxPoints	Maximum points in a non-composite glyph.
in	maxContours	Maximum contours in a non-composite glyph.

Definition at line 1954 of file [hex2otf.c](#).

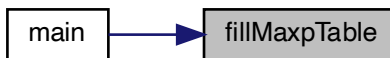
```

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:



5.1.5.27 fillNameTable()

```
void fillNameTable (  
    Font * font,  
    NameStrings nameStrings )
```

Fill a "name" font table.

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

Parameters

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

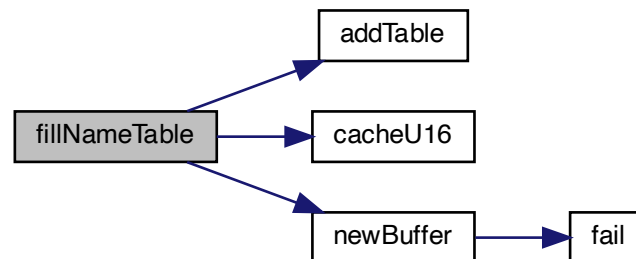
Definition at line 2366 of file [hex2otf.c](#).

```

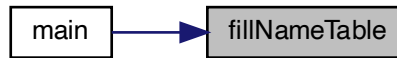
02367 {
02368     Buffer *name = newBuffer (2048);
02369     addTable (font, "name", name);
02370     size_t nameStringCount = 0;
02371     for (size_t i = 0; i < MAX_NAME_IDS; i++)
02372         nameStringCount += !nameStrings[i];
02373     cacheU16 (name, 0); // version
02374     cacheU16 (name, nameStringCount); // 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:



5.1.5.28 fillOS2Table()

```
void fillOS2Table (
    Font * font )
```

Fill an "OS/2" font table.

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

Parameters

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

Definition at line [1986](#) of file [hex2otf.c](#).

```

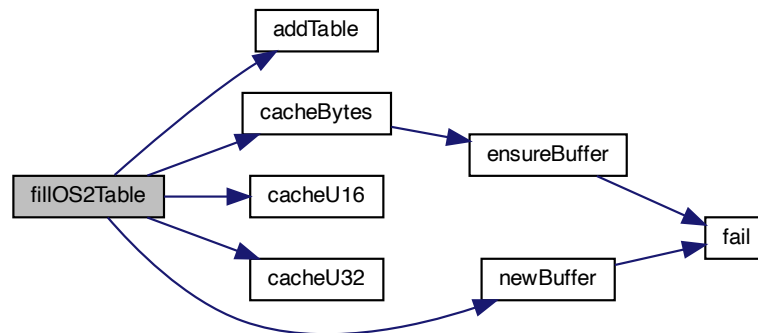
01987 {
01988     Buffer *os2 = newBuffer (100);
01989     addTable (font, "OS/2", os2);
01990     cacheU16 (os2, 5); // version
01991     // HACK: Average glyph width is not actually calculated.
01992     cacheU16 (os2, FU (font->maxWidth)); // xAvgCharWidth
01993     cacheU16 (os2, 400); // usWeightClass = Normal
01994     cacheU16 (os2, 5); // usWidthClass = Medium
01995     const uint\_fast16\_t typeFlags =
01996         + B0 (0) // reserved
01997         // usage permissions, one of:
01998         // Default: Installable embedding
01999         + B0 (1) // Restricted License embedding
02000         + B0 (2) // Preview & Print embedding
02001         + B0 (3) // Editable embedding
02002         // 4-7 reserved
02003         + B0 (8) // no subsetting
02004         + B0 (9) // bitmap embedding only
02005         // 10-15 reserved
02006     ;
02007     cacheU16 (os2, typeFlags); // fsType
02008     cacheU16 (os2, FU (5)); // ySubscriptXSize
02009     cacheU16 (os2, FU (7)); // ySubscriptYSize
  
```

```

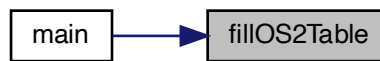
02010 cacheU16 (os2, FU (0)); // ySubscriptXOffset
02011 cacheU16 (os2, FU (1)); // ySubscriptYOffset
02012 cacheU16 (os2, FU (5)); // ySuperscriptXSize
02013 cacheU16 (os2, FU (7)); // ySuperscriptYSize
02014 cacheU16 (os2, FU (0)); // ySuperscriptXOffset
02015 cacheU16 (os2, FU (4)); // ySuperscriptYOffset
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:



5.1.5.29 fillPostTable()

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

Fill a "post" font table.

The "post" table contains information for PostScript printers.

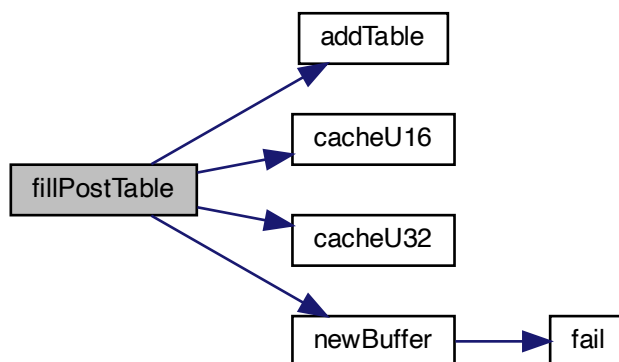
Parameters

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

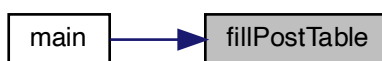
Definition at line 2218 of file [hex2otf.c](#).

```
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:



5.1.5.30 fillTrueType()

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

Add a TrueType table to a font.

Parameters

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

Definition at line 1597 of file [hex2otf.c](#).

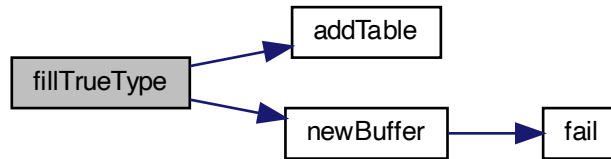
```
01599 {
01600     Buffer *glyph = newBuffer (65536);
01601     addTable (font, "glyph", glyph);
01602     Buffer *loca = newBuffer (4 * (font->glyphCount + 1));
01603     addTable (font, "loca", loca);
01604     *format = LOCA_OFFSET32;
01605     Buffer *endPoints = newBuffer (256);
01606     Buffer *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 (glyph));
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 (glyph, contourCount); // numberOfContours
01669     cacheU16 (glyph, FU (glyph->pos + xMin)); // xMin
01670     cacheU16 (glyph, FU (yMin)); // yMin
01671     cacheU16 (glyph, FU (glyph->pos + xMax)); // xMax
01672     cacheU16 (glyph, FU (yMax)); // yMax
01673     cacheBuffer (glyph, endPoints); // endPtsOfContours[]
01674     cacheU16 (glyph, 0); // instructionLength
01675     cacheBuffer (glyph, flags); // flags[]
01676     cacheBuffer (glyph, xs); // xCoordinates[]
01677     cacheBuffer (glyph, ys); // yCoordinates[]
01678     if (pointCount > *maxPoints)
01679         *maxPoints = pointCount;
01680     if (contourCount > *maxContours)
01681         *maxContours = contourCount;
01682 }
01683 cacheU32 (loca, countBufferedBytes (glyph));
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:



5.1.5.31 freeBuffer()

```
void freeBuffer (
    Buffer * buf )
```

Free the memory previously allocated for a buffer.

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

Parameters

in	buf	The pointer to an array of type <code>Buffer *</code> .

Definition at line 337 of file [hex2otf.c](#).

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

5.1.5.32 initBuffers()

```
void initBuffers (
    size_t count )
```

Initialize an array of buffer pointers to all zeroes.

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

Parameters

in	count	The number of buffer array pointers to allocate.

Definition at line 152 of file [hex2otf.c](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.5.33 main()

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

The main function.

Parameters

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

Returns

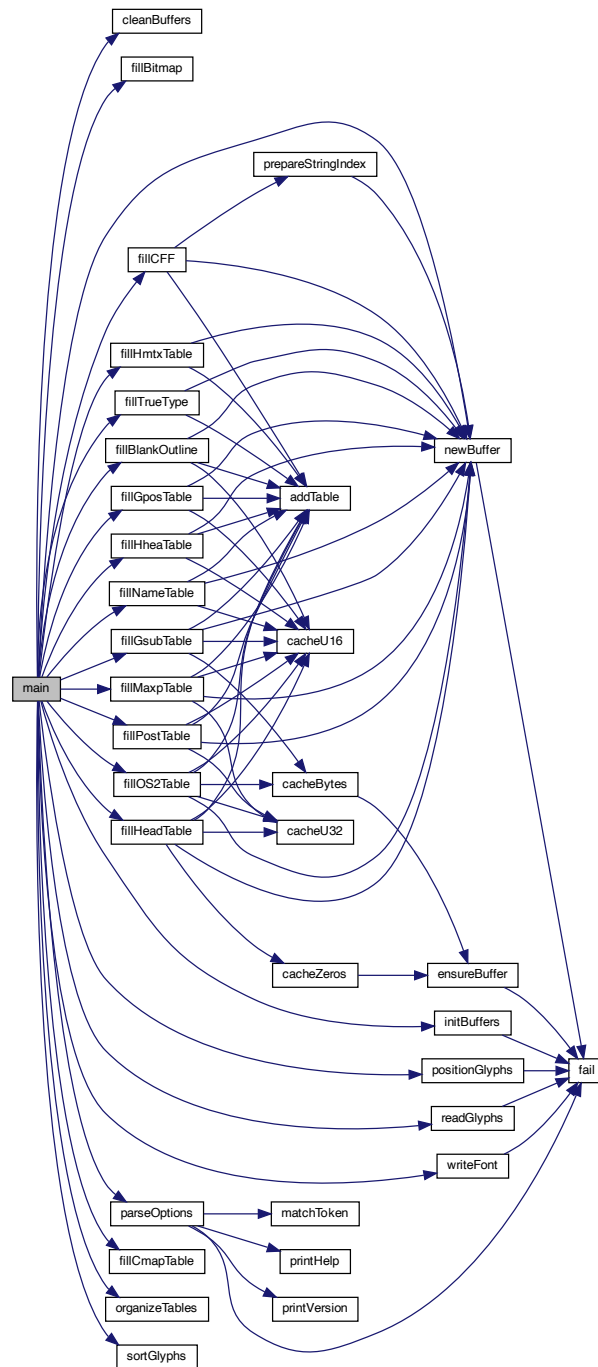
EXIT_FAILURE upon fatal error, EXIT_SUCCESS otherwise.

Definition at line 2603 of file [hex2otf.c](#).

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

```
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:



5.1.5.34 matchToken()

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

Match a command line option with its key for enabling.

Parameters

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

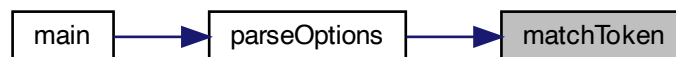
Returns

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:



5.1.5.35 newBuffer()

```
Buffer * newBuffer (
    size_t initialCapacity )
```

Create a new buffer.

This function creates a new buffer array of type [Buffer](#), with an initial size of initialCapacity elements.

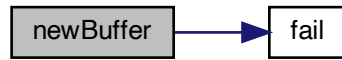
Parameters

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

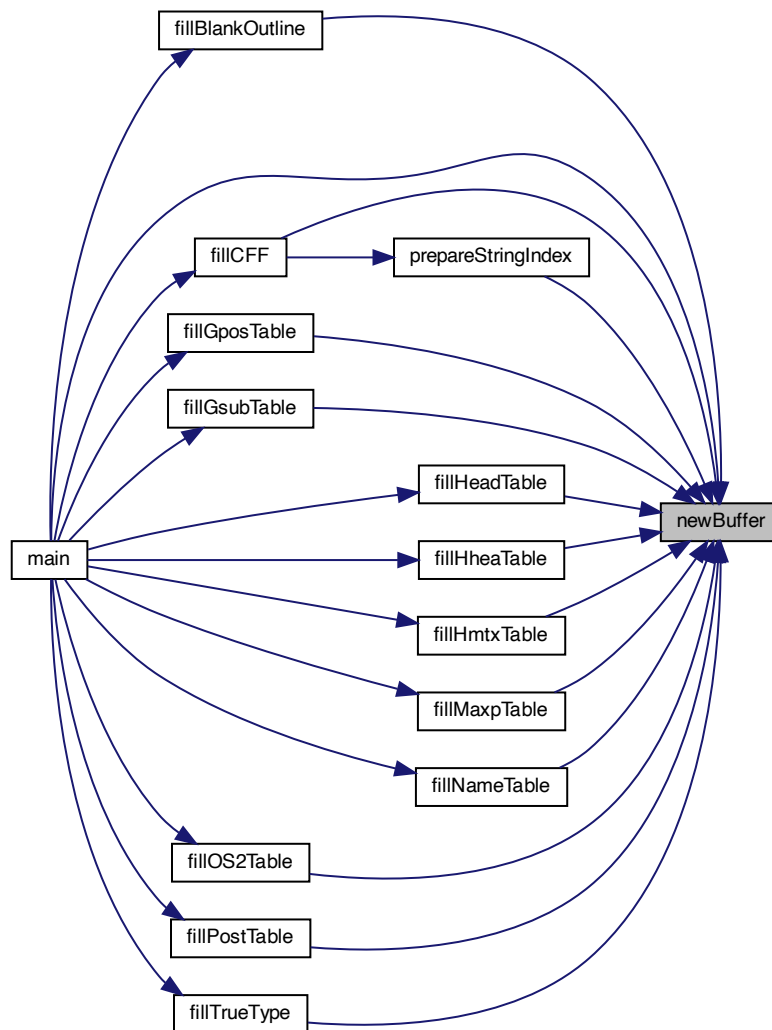
Definition at line 188 of file [hex2otf.c](#).

```
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:



5.1.5.36 organizeTables()

```
void organizeTables (
    Font * font,
    bool isCFF )
```

Sort tables according to OpenType recommendations.

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

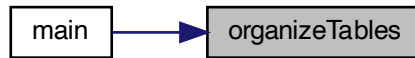
Parameters

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

Definition at line 711 of file [hex2otf.c](#).

```
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:



5.1.5.37 parseOptions()

Options parseOptions (
 char *const argv[const])

Parse command line options.

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

Parameters

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

Returns

Data structure to hold requested command line options.

Definition at line [2500](#) of file [hex2otf.c](#).

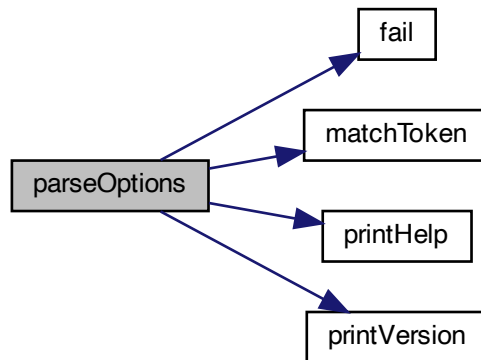
```

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;

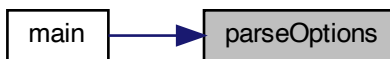
```

```
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:



5.1.5.38 positionGlyphs()

```
void positionGlyphs (  
    Font * font,
```

```
const char * fileName,
pixels_t * xMin )
```

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

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

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

Parameters

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

Definition at line 1061 of file [hex2otf.c](#).

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

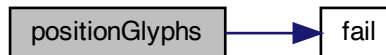


```

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:



5.1.5.39 prepareOffsets()

```

void prepareOffsets (
    size_t * sizes )

```

Prepare 32-bit glyph offsets in a font table.

Parameters

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

Definition at line 1275 of file [hex2otf.c](#).

```

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

Here is the call graph for this function:



5.1.5.40 prepareStringIndex()

```

Buffer * prepareStringIndex (
    const NameStrings names )
```

Prepare a font name string index.

Parameters

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

Returns

Pointer to a [Buffer](#) struct containing the string names.

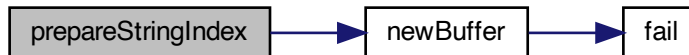
Get the number of elements in array `char *strings[]`.

Definition at line 1291 of file `hex2otf.c`.

```

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 > 0xfffff);
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 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.5.41 printHelp()

```
void printHelp ( )
```

Print help message to stdout and then exit.

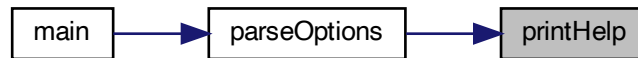
Print help message if invoked with the "--help" option, and then exit successfully.

Definition at line 2426 of file [hex2otf.c](#).

```

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

Here is the caller graph for this function:



5.1.5.42 printVersion()

```
void printVersion ( )
```

Print program version string on stdout.

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

Definition at line 2407 of file [hex2otf.c](#).

```

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

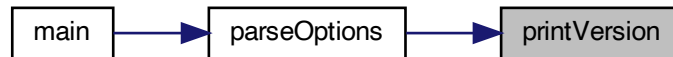
```

```

02415
02416     exit (EXIT_SUCCESS);
02417 }

```

Here is the caller graph for this function:



5.1.5.43 readCodePoint()

```

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

```

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

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

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

Parameters

out	codePoint	The Uni-code 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 }
```

5.1.5.44 readGlyphs()

```

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

Read glyph definitions from a Unifont .hex format file.

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

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

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

Parameters

in,out	font	The font data structure to update with new glyphs.
in	fileName	The name of the Uni-font .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+FFFF
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))

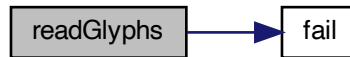
```

```

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:



5.1.5.45 sortGlyphs()

```

void sortGlyphs (
    Font * font )

```

Sort the glyphs in a font by Unicode code point.

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

Parameters

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

Definition at line 1119 of file [hex2otf.c](#).

```

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 caller graph for this function:



5.1.5.46 writeBytes()

```

void writeBytes (
    const byte bytes[],
    size\_t count,
    FILE * file )
```

Write an array of bytes to an output file.

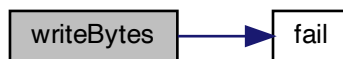
Parameters

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

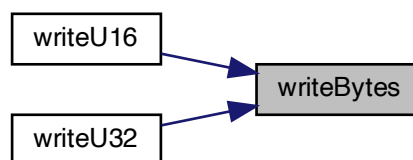
Definition at line 538 of file [hex2otf.c](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.5.47 writeFont()

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

Write OpenType font to output file.

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

Parameters

in	font	Pointer to the font, of type <code>Font</code> *.
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 (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 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.5.48 writeU16()

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

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

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

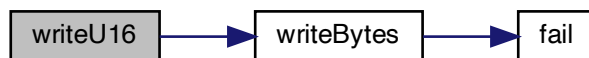
Parameters

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

Definition at line 554 of file [hex2otf.c](#).

```
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:



5.1.5.49 writeU32()

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

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

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

Parameters

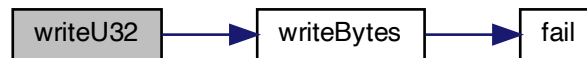
in	value	The 32-bit value to write.
in	file	The file pointer for writing, of type FILE *.

Definition at line 574 of file [hex2otf.c](#).

```
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:



5.1.6 Variable Documentation

5.1.6.1 allBuffers

[Buffer](#)* allBuffers

Initial allocation of empty array of buffer pointers.

Definition at line 139 of file [hex2otf.c](#).

5.1.6.2 bufferCount

size_t bufferCount

Number of buffers in a [Buffer](#) * array.

Definition at line 140 of file [hex2otf.c](#).

5.1.6.3 nextBufferIndex

size_t nextBufferIndex

Index number to tail element of [Buffer](#) * array.

Definition at line 141 of file [hex2otf.c](#).

5.2 hex2otf.c

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

```

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+%.*"PRIFAST32 ///< 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 ()
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 (isdigit (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     }

```

```

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 contor 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 > 0xffffffff);
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  * @brief Fill OpenType bitmap data and location tables.
01717  *
01718  * This function fills an Embedded Bitmap Data (EBDT) Table
01719  * and an Embedded Bitmap Location (EBLC) Table with glyph
01720  * bitmap information. These tables enable embedding bitmaps
01721  * in OpenType fonts. No Embedded Bitmap Scaling (EBS) table
01722  * is used for the bitmap glyphs, only EBDT and EBLC.
01723  *
01724  * @param[in,out] font Pointer to a Font struct in which to add bitmaps.
01725  */
01726 void
01727 fillBitmap (Font *font)
01728 {
01729     const Glyph *const glyphs = getBufferHead (font->glyphs);
01730     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
01731     size_t bitmapsSize = 0;
01732     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
01733         bitmapsSize += glyph->byteCount;
01734     Buffer *ebdt = newBuffer (4 + bitmapsSize);
01735     addTable (font, "EBDT", ebdt);
01736     cacheU16 (ebdt, 2); // majorVersion
01737     cacheU16 (ebdt, 0); // minorVersion
01738     uint_fast8_t byteCount = 0; // unequal to any glyph
01739     pixels_t pos = 0;
01740     bool combining = false;
01741     Buffer *rangeHeads = newBuffer (32);
01742     Buffer *offsets = newBuffer (64);
01743     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
01744     {
01745         if (glyph->byteCount != byteCount || glyph->pos != pos ||
01746             glyph->combining != combining)
01747         {
01748             storeU16 (rangeHeads, glyph - glyphs);
01749             storeU32 (offsets, countBufferedBytes (ebdt));
01750             byteCount = glyph->byteCount;
01751             pos = glyph->pos;
01752             combining = glyph->combining;
01753         }
01754         cacheBytes (ebdt, glyph->bitmap, byteCount);
01755     }
01756     const uint_least16_t *ranges = getBufferHead (rangeHeads);
01757     const uint_least16_t *rangesEnd = getBufferTail (rangeHeads);
01758     uint_fast32_t rangeCount = rangesEnd - ranges;
01759     storeU16 (rangeHeads, font->glyphCount);
01760     Buffer *eblc = newBuffer (4096);
01761     addTable (font, "EBLC", eblc);
01762     cacheU16 (eblc, 2); // majorVersion
01763     cacheU16 (eblc, 0); // minorVersion
01764     cacheU32 (eblc, 1); // numSizes
01765     { // bitmapSizes[0]
01766         cacheU32 (eblc, 56); // indexSubTableArrayOffset
01767         cacheU32 (eblc, (8 + 20) * rangeCount); // indexTablesSize
01768         cacheU32 (eblc, rangeCount); // numberOfIndexSubTables
01769         cacheU32 (eblc, 0); // colorRef
01770         { // hori
01771             cacheU8 (eblc, ASCENDER); // ascender
01772             cacheU8 (eblc, -DESCENDER); // descender
01773             cacheU8 (eblc, font->maxWidth); // widthMax
01774             cacheU8 (eblc, 1); // caretSlopeNumerator
01775             cacheU8 (eblc, 0); // caretSlopeDenominator
01776             cacheU8 (eblc, 0); // caretOffset
01777         }
01778     }
01779 }

```

```

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     }

```

```

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 () {
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 () {
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)

```

```

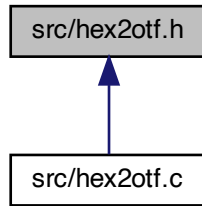
02588     fail ("At most one outline format can be accepted.");
02589     if (!(cff || cff2 || opt.truetype || opt.bitmap))
02590         fail ("Invalid format.");
02591     opt.cff = cff + cff2 * 2;
02592     return opt;
02593 }
02594 /**
02595  @brief The main function.
02596  @param[in] argc The number 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 }

```

5.3 src/hex2otf.h File Reference

[hex2otf.h](#) - Header file for [hex2otf.c](#)

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



Data Structures

- struct [NamePair](#)
Data structure for a font ID number and name character string.

Macros

- `#define UNIFONT_VERSION "15.0.02"`
Current Unifont version.
- `#define DEFAULT_ID0 "Copyright © 1998-2022 Roman Czyborra, Paul Hardy, \Qianqian Fang, Andrew Miller, Johnnie Weaver, David Corbett, \Nils Moskopp, Rebecca Bettencourt, et al."`
- `#define DEFAULT_ID1 "Unifont"`
Default NameID 1 string ([Font](#) Family)
- `#define DEFAULT_ID2 "Regular"`
Default NameID 2 string ([Font](#) Subfamily)
- `#define DEFAULT_ID5 "Version "UNIFONT_VERSION`
Default NameID 5 string (Version of the Name [Table](#))
- `#define DEFAULT_ID11 "https://unifoundry.com/unifont/"`
Default NameID 11 string ([Font](#) Vendor URL)
- `#define DEFAULT_ID13 "Dual license: SIL Open Font License version 1.1, \and GNU GPL version 2 or later with the GNU Font Embedding Exception."`
Default NameID 13 string (License Description)
- `#define DEFAULT_ID14 "http://unifoundry.com/LICENSE.txt, \https://scripts.sil.org/OFL"`
Default NameID 14 string (License Information URLs)
- `#define NAMEPAIR(n) {(n), DEFAULT_ID##n}`
Macro to initialize name identifier codes to default values defined above.

Typedefs

- typedef struct [NamePair](#) NamePair
Data structure for a font ID number and name character string.

Variables

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

5.3.1 Detailed Description

[hex2otf.h](#) - Header file for [hex2otf.c](#)

Copyright

Copyright © 2022 何志翔 (He Zhixiang)

Author

何志翔 (He Zhixiang)

Definition in file [hex2otf.h](#).

5.3.2 Macro Definition Documentation

5.3.2.1 DEFAULT_ID0

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

Define default strings for some TrueType font NameID strings.

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

Default NameID 0 string (Copyright Notice)

Definition at line 53 of file [hex2otf.h](#).

5.3.2.2 DEFAULT_ID1

```
#define DEFAULT_ID1 "Unifont"
```

Default NameID 1 string ([Font](#) Family)

Definition at line 57 of file [hex2otf.h](#).

5.3.2.3 DEFAULT_ID11

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

Default NameID 11 string ([Font](#) Vendor URL)

Definition at line 64 of file [hex2otf.h](#).

5.3.2.4 DEFAULT_ID13

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

Default NameID 13 string (License Description)

Definition at line 67 of file [hex2otf.h](#).

5.3.2.5 DEFAULT_ID14

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

Default NameID 14 string (License Information URLs)

Definition at line 71 of file [hex2otf.h](#).

5.3.2.6 DEFAULT_ID2

```
#define DEFAULT_ID2 "Regular"
```

Default NameID 2 string ([Font](#) Subfamily)

Definition at line 58 of file [hex2otf.h](#).

5.3.2.7 DEFAULT_ID5

```
#define DEFAULT_ID5 "Version "UNIFONT_VERSION
```

Default NameID 5 string (Version of the Name [Table](#))

Definition at line [61](#) of file [hex2otf.h](#).

5.3.2.8 NAMEPAIR

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

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

Definition at line [84](#) of file [hex2otf.h](#).

5.3.2.9 UNIFONT_VERSION

```
#define UNIFONT_VERSION "15.0.02"
```

Current Unifont version.

Definition at line [36](#) of file [hex2otf.h](#).

5.3.3 Variable Documentation

5.3.3.1 defaultNames

```
const NamePair defaultNames[]
```

Initial value:

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

Allocate array of NameID codes with default values.

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

Definition at line [93](#) of file [hex2otf.h](#).

5.4 hex2otf.h

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

```

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 "15.0.02" ///< 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

```

5.5 src/unibdf2hex.c File Reference

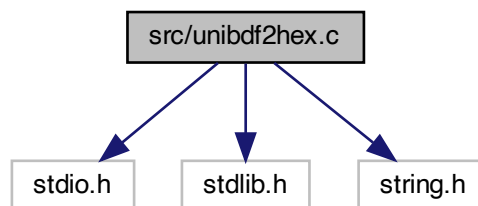
unibdf2hex - Convert a BDF file into a unifont.hex file

```

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

```

Include dependency graph for unibdf2hex.c:



Macros

- #define **UNISTART** 0x3400
First Unicode code point to examine.

- `#define UNISTOP 0x4DBF`
Last Unicode code point to examine.
- `#define MAXBUF 256`
Maximum allowable input file line length - 1.

Functions

- `int main ()`
The main function.

5.5.1 Detailed Description

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

Author

Paul Hardy, January 2008

Copyright

Copyright (C) 2008, 2013 Paul Hardy

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

Definition in file [unibdf2hex.c](#).

5.5.2 Macro Definition Documentation

5.5.2.1 MAXBUF

`#define MAXBUF 256`

Maximum allowable input file line length - 1.

Definition at line [37](#) of file [unibdf2hex.c](#).

5.5.2.2 UNISTART

```
#define UNISTART 0x3400
```

First Unicode code point to examine.

Definition at line 34 of file [unibdf2hex.c](#).

5.5.2.3 UNISTOP

```
#define UNISTOP 0x4DBF
```

Last Unicode code point to examine.

Definition at line 35 of file [unibdf2hex.c](#).

5.5.3 Function Documentation

5.5.3.1 main()

```
int main ( )
```

The main function.

Returns

Exit status is always 0 (successful termination).

Definition at line 46 of file [unibdf2hex.c](#).

```
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 }

```

5.6 unbdf2hex.c

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

```

00001 /**
00002  @file unbdf2hex.c
00003
00004  @brief unbdf2hex - 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()
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         }

```

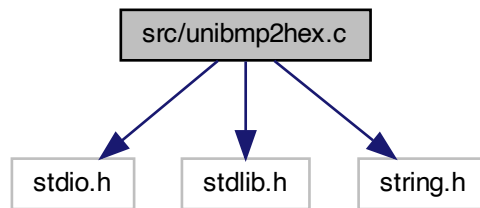
```
00110  }  
00111  exit (0);  
00112 }
```

5.7 src/unibmp2hex.c File Reference

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

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

Include dependency graph for unibmp2hex.c:



Macros

- `#define` `MAXBUF` 256
Maximum input file line length - 1.

Functions

- `int` `main` (`int` argc, `char` *argv[])
The main function.

Variables

- `unsigned` `hexdigit` [16][4]
32 bit representation of 16x8 0..F bitmap
- `unsigned` `uniplane` =0
Unicode plane number, 0..0xff ff.
- `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
- struct {
 - char `filetype` [2]
 - int `file_size`
 - int `image_offset`
 - int `info_size`
 - int `width`
 - int `height`
 - int `nplanes`
 - int `bits_per_pixel`
 - int `compression`
 - int `image_size`
 - int `x_ppm`
 - int `y_ppm`
 - int `ncolors`
 - int `important_colors`
- } `bmp_header`
- unsigned char `color_table` [256][4]

5.7.1 Detailed Description

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

Author

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

Copyright

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

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

Definition in file [unibmp2hex.c](#).

5.7.2 Macro Definition Documentation

5.7.2.1 MAXBUF

```
#define MAXBUF 256
```

Maximum input file line length - 1.

Definition at line [104](#) of file [unibmp2hex.c](#).

5.7.3 Function Documentation

5.7.3.1 main()

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

The main function.

Parameters

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

Returns

This program exits with status 0.

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

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

```

00171     4 = force glyph to quadruple-width.
00172 */
00173 char wide[0x200000]={0x200000 * 0};
00174
00175 char *infile="", *outfile=""; /* names of input and output files */
00176 FILE *infp, *outfp; /* file pointers of input and output files */
00177
00178 if (argc > 1) {
00179     for (i = 1; i < argc; i++) {
00180         if (argv[i][0] == '-') { /* this is an option argument */
00181             switch (argv[i][1]) {
00182                 case 'i': /* name of input file */
00183                     infile = &argv[i][2];
00184                     break;
00185                 case 'o': /* name of output file */
00186                     outfile = &argv[i][2];
00187                     break;
00188                 case 'p': /* specify a Unicode plane */
00189                     sscanf (&argv[i][2], "%x", &uniplane); /* Get Unicode plane */
00190                     planeset = 1; /* Use specified range, not what's in bitmap */
00191                     break;
00192                 case 'w': /* force wide (16 pixels) for each glyph */
00193                     forcewide = 1;
00194                     break;
00195                 default: /* if unrecognized option, print list and exit */
00196                     fprintf (stderr, "\nSyntax:\n\n");
00197                     fprintf (stderr, " %s -p<Unicode_Page> ", argv[0]);
00198                     fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
00199                     fprintf (stderr, " -w specifies .wbmp output instead of ");
00200                     fprintf (stderr, "default Windows .bmp output.\n\n");
00201                     fprintf (stderr, " -p is followed by 1 to 6 ");
00202                     fprintf (stderr, "Unicode plane hex digits ");
00203                     fprintf (stderr, "(default is Page 0).\n\n");
00204                     fprintf (stderr, "\nExample:\n\n");
00205                     fprintf (stderr, " %s -p83 -iunifont.hex -ou83.bmp\n\n",
00206                             argv[0]);
00207                     exit (1);
00208             }
00209         }
00210     }
00211 }
00212 /*
00213 Make sure we can open any I/O files that were specified before
00214 doing anything else.
00215 */
00216 if (strlen (infile) > 0) {
00217     if ((infp = fopen (infile, "r")) == NULL) {
00218         fprintf (stderr, "Error: can't open %s for input.\n", infile);
00219         exit (1);
00220     }
00221 }
00222 else {
00223     infp = stdin;
00224 }
00225 if (strlen (outfile) > 0) {
00226     if ((outfp = fopen (outfile, "w")) == NULL) {
00227         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00228         exit (1);
00229     }
00230 }
00231 else {
00232     outfp = stdout;
00233 }
00234 /*
00235 Initialize selected code points for double width (16x16).
00236 Double-width is forced in cases where a glyph (usually a combining
00237 glyph) only occupies the left-hand side of a 16x16 grid, but must
00238 be rendered as double-width to appear properly with other glyphs
00239 in a given script. If additions were made to a script after
00240 Unicode 5.0, the Unicode version is given in parentheses after
00241 the script name.
00242 */
00243 for (i = 0x0700; i <= 0x074F; i++) wide[i] = 1; /* Syriac */
00244 for (i = 0x0800; i <= 0x083F; i++) wide[i] = 1; /* Samaritan (5.2) */
00245 for (i = 0x0900; i <= 0x0DFF; i++) wide[i] = 1; /* Indic */
00246 for (i = 0x1000; i <= 0x109F; i++) wide[i] = 1; /* Myanmar */
00247 for (i = 0x1100; i <= 0x11FF; i++) wide[i] = 1; /* Hangul Jamo */
00248 for (i = 0x1400; i <= 0x167F; i++) wide[i] = 1; /* Canadian Aboriginal */
00249 for (i = 0x1700; i <= 0x171F; i++) wide[i] = 1; /* Tagalog */
00250 for (i = 0x1720; i <= 0x173F; i++) wide[i] = 1; /* Hanunoo */
00251 for (i = 0x1740; i <= 0x175F; i++) wide[i] = 1; /* Buhid */

```

```

00252 for (i = 0x1760; i <= 0x177F; i++) wide[i] = 1; /* Tagbanwa */
00253 for (i = 0x1780; i <= 0x17FF; i++) wide[i] = 1; /* Khmer */
00254 for (i = 0x18B0; i <= 0x18FF; i++) wide[i] = 1; /* Ext. Can. Aboriginal */
00255 for (i = 0x1800; i <= 0x18AF; i++) wide[i] = 1; /* Mongolian */
00256 for (i = 0x1900; i <= 0x194F; i++) wide[i] = 1; /* Limbu */
00257 // for (i = 0x1980; i <= 0x19DF; i++) wide[i] = 1; /* New Tai Lue */
00258 for (i = 0x1A00; i <= 0x1A1F; i++) wide[i] = 1; /* Buginese */
00259 for (i = 0x1A20; i <= 0x1AAF; i++) wide[i] = 1; /* Tai Tham (5.2) */
00260 for (i = 0x1B00; i <= 0x1B7F; i++) wide[i] = 1; /* Balinese */
00261 for (i = 0x1B80; i <= 0x1BBF; i++) wide[i] = 1; /* Sundanese (5.1) */
00262 for (i = 0x1BC0; i <= 0x1BFF; i++) wide[i] = 1; /* Batak (6.0) */
00263 for (i = 0x1C00; i <= 0x1C4F; i++) wide[i] = 1; /* Lepcha (5.1) */
00264 for (i = 0x1CC0; i <= 0x1CCF; i++) wide[i] = 1; /* Sundanese Supplement */
00265 for (i = 0x1CD0; i <= 0x1CFF; i++) wide[i] = 1; /* Vedic Extensions (5.2) */
00266 wide[0x2329] = wide[0x232A] = 1; /* Left- & Right-pointing Angle Brackets */
00267 for (i = 0x2E80; i <= 0xA4CF; i++) wide[i] = 1; /* CJK */
00268 // for (i = 0x9FD8; i <= 0x9FE9; i++) wide[i] = 4; /* CJK quadruple-width */
00269 for (i = 0xA900; i <= 0xA92F; i++) wide[i] = 1; /* Kayah Li (5.1) */
00270 for (i = 0xA930; i <= 0xA95F; i++) wide[i] = 1; /* Rejang (5.1) */
00271 for (i = 0xA960; i <= 0xA97F; i++) wide[i] = 1; /* Hangul Jamo Extended-A */
00272 for (i = 0xA980; i <= 0xA9DF; i++) wide[i] = 1; /* Javanese (5.2) */
00273 for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham (5.1) */
00274 for (i = 0xA9E0; i <= 0xA9FF; i++) wide[i] = 1; /* Myanmar Extended-B */
00275 for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham */
00276 for (i = 0xAA60; i <= 0xAA7F; i++) wide[i] = 1; /* Myanmar Extended-A */
00277 for (i = 0xAAE0; i <= 0xAADF; i++) wide[i] = 1; /* Meetei Mayek Ext (6.0) */
00278 for (i = 0xABC0; i <= 0xABFF; i++) wide[i] = 1; /* Meetei Mayek (5.2) */
00279 for (i = 0xAC00; i <= 0xD7AF; i++) wide[i] = 1; /* Hangul Syllables */
00280 for (i = 0xD7B0; i <= 0xD7FF; i++) wide[i] = 1; /* Hangul Jamo Extended-B */
00281 for (i = 0xF900; i <= 0xFAFF; i++) wide[i] = 1; /* CJK Compatibility */
00282 for (i = 0xFE10; i <= 0xFE1F; i++) wide[i] = 1; /* Vertical Forms */
00283 for (i = 0xFE30; i <= 0xFE60; i++) wide[i] = 1; /* CJK Compatibility Forms */
00284 for (i = 0xFFE0; i <= 0xFFE6; i++) wide[i] = 1; /* CJK Compatibility Forms */
00285
00286 wide[0x303F] = 0; /* CJK half-space fill */
00287
00288 /* Supplemental Multilingual Plane (Plane 01) */
00289 for (i = 0x010A00; i <= 0x010A5F; i++) wide[i] = 1; /* Kharoshthi */
00290 for (i = 0x011000; i <= 0x01107F; i++) wide[i] = 1; /* Brahmi */
00291 for (i = 0x011080; i <= 0x0110CF; i++) wide[i] = 1; /* Kaithi */
00292 for (i = 0x011100; i <= 0x01114F; i++) wide[i] = 1; /* Chakma */
00293 for (i = 0x011180; i <= 0x0111DF; i++) wide[i] = 1; /* Sharada */
00294 for (i = 0x011200; i <= 0x01124F; i++) wide[i] = 1; /* Khojki */
00295 for (i = 0x0112B0; i <= 0x0112FF; i++) wide[i] = 1; /* Khudawadi */
00296 for (i = 0x011300; i <= 0x01137F; i++) wide[i] = 1; /* Grantha */
00297 for (i = 0x011400; i <= 0x01147F; i++) wide[i] = 1; /* Newa */
00298 for (i = 0x011480; i <= 0x0114DF; i++) wide[i] = 1; /* Tirhuta */
00299 for (i = 0x011580; i <= 0x0115FF; i++) wide[i] = 1; /* Siddham */
00300 for (i = 0x011600; i <= 0x01165F; i++) wide[i] = 1; /* Modi */
00301 for (i = 0x011660; i <= 0x01167F; i++) wide[i] = 1; /* Mongolian Suppl. */
00302 for (i = 0x011680; i <= 0x0116CF; i++) wide[i] = 1; /* Takri */
00303 for (i = 0x011700; i <= 0x01173F; i++) wide[i] = 1; /* Ahom */
00304 for (i = 0x011800; i <= 0x01184F; i++) wide[i] = 1; /* Dogra */
00305 for (i = 0x011900; i <= 0x01195F; i++) wide[i] = 1; /* Dives Akuru */
00306 for (i = 0x0119A0; i <= 0x0119FF; i++) wide[i] = 1; /* Nandinagari */
00307 for (i = 0x011A00; i <= 0x011A4F; i++) wide[i] = 1; /* Zanabazar Square */
00308 for (i = 0x011A50; i <= 0x011AAF; i++) wide[i] = 1; /* Soyombo */
00309 for (i = 0x011B00; i <= 0x011B5F; i++) wide[i] = 1; /* Devanagari Extended-A */
00310 for (i = 0x011F00; i <= 0x011F5F; i++) wide[i] = 1; /* Kawi */
00311 for (i = 0x011C00; i <= 0x011C6F; i++) wide[i] = 1; /* Bhaiksuki */
00312 for (i = 0x011C70; i <= 0x011CBF; i++) wide[i] = 1; /* Marchen */
00313 for (i = 0x011D00; i <= 0x011D5F; i++) wide[i] = 1; /* Masaram Gondi */
00314 for (i = 0x011EE0; i <= 0x011EFF; i++) wide[i] = 1; /* Makasar */
00315 for (i = 0x012F90; i <= 0x012FFF; i++) wide[i] = 1; /* Cypro-Minoan */
00316 /* Make Bassa Vah all single width or all double width */
00317 for (i = 0x016AD0; i <= 0x016AFF; i++) wide[i] = 1; /* Bassa Vah */
00318 for (i = 0x016B00; i <= 0x016B8F; i++) wide[i] = 1; /* Pahawh Hmong */
00319 for (i = 0x016F00; i <= 0x016F9F; i++) wide[i] = 1; /* Miao */
00320 for (i = 0x016FE0; i <= 0x016FFF; i++) wide[i] = 1; /* Ideograph Sym/Punct */
00321 for (i = 0x017000; i <= 0x0187FF; i++) wide[i] = 1; /* Tangut */
00322 for (i = 0x018800; i <= 0x018AFF; i++) wide[i] = 1; /* Tangut Components */
00323 for (i = 0x01AFF0; i <= 0x01AFFF; i++) wide[i] = 1; /* Kana Extended-B */
00324 for (i = 0x01B000; i <= 0x01B0FF; i++) wide[i] = 1; /* Kana Supplement */
00325 for (i = 0x01B100; i <= 0x01B12F; i++) wide[i] = 1; /* Kana Extended-A */
00326 for (i = 0x01B170; i <= 0x01B2FF; i++) wide[i] = 1; /* Nushu */
00327 for (i = 0x01CF00; i <= 0x01CFCF; i++) wide[i] = 1; /* Znamenny Musical */
00328 for (i = 0x01D100; i <= 0x01D1FF; i++) wide[i] = 1; /* Musical Symbols */
00329 for (i = 0x01D800; i <= 0x01DAAF; i++) wide[i] = 1; /* Sutton SignWriting */
00330 for (i = 0x01E2C0; i <= 0x01E2FF; i++) wide[i] = 1; /* Wancho */
00331 for (i = 0x01E800; i <= 0x01E8DF; i++) wide[i] = 1; /* Mende Kikakui */
00332 for (i = 0x01F200; i <= 0x01F2FF; i++) wide[i] = 1; /* Encl Ideograp Suppl */

```

```

00333 wide[0x01F5E7] = 1;                                /* Three Rays Right */
00334
00335 /*
00336 Determine whether or not the file is a Microsoft Windows Bitmap file.
00337 If it starts with 'B', 'M', assume it's a Windows Bitmap file.
00338 Otherwise, assume it's a Wireless Bitmap file.
00339
00340 WARNING: There isn't much in the way of error checking here --
00341 if you give it a file that wasn't first created by hex2bmp.c,
00342 all bets are off.
00343 */
00344 fatal = 0; /* assume everything is okay with reading input file */
00345 if ((header[0] = fgetc (infp)) != EOF) {
00346     if ((header[1] = fgetc (infp)) != EOF) {
00347         if (header[0] == 'B' && header[1] == 'M') {
00348             wbmp = 0; /* Not a Wireless Bitmap -- it's a Windows Bitmap */
00349         }
00350         else {
00351             wbmp = 1; /* Assume it's a Wireless Bitmap */
00352         }
00353     }
00354     else
00355         fatal = 1;
00356 }
00357 else
00358     fatal = 1;
00359
00360 if (fatal) {
00361     fprintf (stderr, "Fatal error; end of input file.\n\n");
00362     exit (1);
00363 }
00364 /*
00365 If this is a Wireless Bitmap (.wbmp) format file,
00366 skip the header and point to the start of the bitmap itself.
00367 */
00368 if (wbmp) {
00369     for (i=2; i<6; i++)
00370         header[i] = fgetc (infp);
00371     /*
00372     Now read the bitmap.
00373     */
00374     for (i=0; i < 32*17; i++) {
00375         for (j=0; j < 32*18/8; j++) {
00376             inchar = fgetc (infp);
00377             bitmap[i][j] = ~inchar; /* invert bits for proper color */
00378         }
00379     }
00380 }
00381 /*
00382 Otherwise, treat this as a Windows Bitmap file, because we checked
00383 that it began with "BM". Save the header contents for future use.
00384 Expect a 14 byte standard BITMAPFILEHEADER format header followed
00385 by a 40 byte standard BITMAPINFOHEADER Device Independent Bitmap
00386 header, with data stored in little-endian format.
00387 */
00388 else {
00389     for (i = 2; i < 54; i++)
00390         header[i] = fgetc (infp);
00391
00392     bmp_header.filetype[0] = 'B';
00393     bmp_header.filetype[1] = 'M';
00394
00395     bmp_header.file_size =
00396         (header[2] & 0xFF) | ((header[3] & 0xFF) « 8) |
00397         ((header[4] & 0xFF) « 16) | ((header[5] & 0xFF) « 24);
00398
00399     /* header bytes 6..9 are reserved */
00400
00401     bmp_header.image_offset =
00402         (header[10] & 0xFF) | ((header[11] & 0xFF) « 8) |
00403         ((header[12] & 0xFF) « 16) | ((header[13] & 0xFF) « 24);
00404
00405     bmp_header.info_size =
00406         (header[14] & 0xFF) | ((header[15] & 0xFF) « 8) |
00407         ((header[16] & 0xFF) « 16) | ((header[17] & 0xFF) « 24);
00408
00409     bmp_header.width =
00410         (header[18] & 0xFF) | ((header[19] & 0xFF) « 8) |
00411         ((header[20] & 0xFF) « 16) | ((header[21] & 0xFF) « 24);
00412
00413     bmp_header.height =

```



```

00414     (header[22] & 0xFF) | ((header[23] & 0xFF) << 8) |
00415     ((header[24] & 0xFF) << 16) | ((header[25] & 0xFF) << 24);
00416
00417     bmp_header.nplanes =
00418     (header[26] & 0xFF) | ((header[27] & 0xFF) << 8);
00419
00420     bmp_header.bits_per_pixel =
00421     (header[28] & 0xFF) | ((header[29] & 0xFF) << 8);
00422
00423     bmp_header.compression =
00424     (header[30] & 0xFF) | ((header[31] & 0xFF) << 8) |
00425     ((header[32] & 0xFF) << 16) | ((header[33] & 0xFF) << 24);
00426
00427     bmp_header.image_size =
00428     (header[34] & 0xFF) | ((header[35] & 0xFF) << 8) |
00429     ((header[36] & 0xFF) << 16) | ((header[37] & 0xFF) << 24);
00430
00431     bmp_header.x_ppm =
00432     (header[38] & 0xFF) | ((header[39] & 0xFF) << 8) |
00433     ((header[40] & 0xFF) << 16) | ((header[41] & 0xFF) << 24);
00434
00435     bmp_header.y_ppm =
00436     (header[42] & 0xFF) | ((header[43] & 0xFF) << 8) |
00437     ((header[44] & 0xFF) << 16) | ((header[45] & 0xFF) << 24);
00438
00439     bmp_header.ncolors =
00440     (header[46] & 0xFF) | ((header[47] & 0xFF) << 8) |
00441     ((header[48] & 0xFF) << 16) | ((header[49] & 0xFF) << 24);
00442
00443     bmp_header.important_colors =
00444     (header[50] & 0xFF) | ((header[51] & 0xFF) << 8) |
00445     ((header[52] & 0xFF) << 16) | ((header[53] & 0xFF) << 24);
00446
00447     if (bmp_header.ncolors == 0)
00448         bmp_header.ncolors = 1 << bmp_header.bits_per_pixel;
00449
00450     /* If a Color Table exists, read it */
00451     if (bmp_header.ncolors > 0 && bmp_header.bits_per_pixel <= 8) {
00452         for (i = 0; i < bmp_header.ncolors; i++) {
00453             color_table[i][0] = fgetc (infp); /* Red */
00454             color_table[i][1] = fgetc (infp); /* Green */
00455             color_table[i][2] = fgetc (infp); /* Blue */
00456             color_table[i][3] = fgetc (infp); /* Alpha */
00457         }
00458         /*
00459          Determine from the first color table entry whether we
00460          are inverting the resulting bitmap image.
00461          */
00462         if ( (color_table[0][0] + color_table[0][1] + color_table[0][2])
00463              < (3 * 128) ) {
00464             color_mask = 0xFF;
00465         }
00466     }
00467
00468 #ifdef DEBUG
00469     /*
00470      Print header info for possibly adding support for
00471      additional file formats in the future, to determine
00472      how the bitmap is encoded.
00473      */
00474     fprintf (stderr, "Filetype: '%c%c'\n",
00475             bmp_header.filetype[0], bmp_header.filetype[1]);
00476     fprintf (stderr, "File Size: %d\n", bmp_header.file_size);
00477     fprintf (stderr, "Image Offset: %d\n", bmp_header.image_offset);
00478     fprintf (stderr, "Info Header Size: %d\n", bmp_header.info_size);
00479     fprintf (stderr, "Image Width: %d\n", bmp_header.width);
00480     fprintf (stderr, "Image Height: %d\n", bmp_header.height);
00481     fprintf (stderr, "Number of Planes: %d\n", bmp_header.nplanes);
00482     fprintf (stderr, "Bits per Pixel: %d\n", bmp_header.bits_per_pixel);
00483     fprintf (stderr, "Compression Method: %d\n", bmp_header.compression);
00484     fprintf (stderr, "Image Size: %d\n", bmp_header.image_size);
00485     fprintf (stderr, "X Pixels per Meter: %d\n", bmp_header.x_ppm);
00486     fprintf (stderr, "Y Pixels per Meter: %d\n", bmp_header.y_ppm);
00487     fprintf (stderr, "Number of Colors: %d\n", bmp_header.ncolors);
00488     fprintf (stderr, "Important Colors: %d\n", bmp_header.important_colors);
00489
00490 #endif
00491
00492     /*
00493      Now read the bitmap.

```

```

00495  */
00496  for (i = 32*17-1; i >= 0; i--) {
00497      for (j=0; j < 32*18/8; j++) {
00498          next_pixels = 0x00; /* initialize next group of 8 pixels */
00499          /* Read a monochrome image -- the original case */
00500          if (bmp_header.bits_per_pixel == 1) {
00501              next_pixels = fgetc (infp);
00502          }
00503          /* Read a 32 bit per pixel RGB image; convert to monochrome */
00504          else if ( bmp_header.bits_per_pixel == 24 ||
00505                   bmp_header.bits_per_pixel == 32) {
00506              next_pixels = 0;
00507              for (k = 0; k < 8; k++) { /* get next 8 pixels */
00508                  this_pixel = (fgetc (infp) & 0xFF) +
00509                              (fgetc (infp) & 0xFF) +
00510                              (fgetc (infp) & 0xFF);
00511
00512                  if (bmp_header.bits_per_pixel == 32) {
00513                      (void) fgetc (infp); /* ignore alpha value */
00514                  }
00515
00516                  /* convert RGB color space to monochrome */
00517                  if (this_pixel >= (128 * 3))
00518                      this_pixel = 0;
00519                  else
00520                      this_pixel = 1;
00521
00522                  /* shift next pixel color into place for 8 pixels total */
00523                  next_pixels = (next_pixels « 1) | this_pixel;
00524              }
00525          }
00526          if (bmp_header.height < 0) { /* Bitmap drawn top to bottom */
00527              bitmap [(32*17-1) - i][j] = next_pixels;
00528          }
00529          else { /* Bitmap drawn bottom to top */
00530              bitmap [i][j] = next_pixels;
00531          }
00532      }
00533  }
00534
00535  /*
00536   If any bits are set in color_mask, apply it to
00537   entire bitmap to invert black <--> white.
00538  */
00539  if (color_mask != 0x00) {
00540      for (i = 32*17-1; i >= 0; i--) {
00541          for (j=0; j < 32*18/8; j++) {
00542              bitmap [i][j] ^= color_mask;
00543          }
00544      }
00545  }
00546
00547  }
00548
00549  /*
00550   We've read the entire file. Now close the input file pointer.
00551  */
00552  fclose (infp);
00553  /*
00554   We now have the header portion in the header[] array,
00555   and have the bitmap portion from top-to-bottom in the bitmap[] array.
00556  */
00557  /*
00558   If no Unicode range (U+nnnnnn00 through U+nnnnnnFF) was specified
00559   with a -p parameter, determine the range from the digits in the
00560   bitmap itself.
00561
00562   Store bitmaps for the hex digit patterns that this file uses.
00563  */
00564  if (!planeset) { /* If Unicode range not specified with -p parameter */
00565      for (i = 0x0; i <= 0xF; i++) { /* hex digit pattern we're storing */
00566          for (j = 0; j < 4; j++) {
00567              hexdigit[i][j] =
00568                  ((unsigned)bitmap[32 * (i+1) + 4 * j + 8][6] « 24) |
00569                  ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 1][6] « 16) |
00570                  ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 2][6] « 8) |
00571                  ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 3][6] );
00572          }
00573      }
00574  }
00575  /*
00576   Read the Unicode plane digits into arrays for comparison, to

```

```

00576     determine the upper four hex digits of the glyph addresses.
00577     */
00578     for (i = 0; i < 4; i++) {
00579         for (j = 0; j < 4; j++) {
00580             unidigit[i][j] =
00581                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 1][i + 3] « 24 ) |
00582                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 2][i + 3] « 16 ) |
00583                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 3][i + 3] « 8 ) |
00584                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 4][i + 3] );
00585         }
00586     }
00587
00588     tmpsum = 0;
00589     for (i = 4; i < 6; i++) {
00590         for (j = 0; j < 4; j++) {
00591             unidigit[i][j] =
00592                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 ][i] « 24 ) |
00593                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 1][i] « 16 ) |
00594                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 2][i] « 8 ) |
00595                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 3][i] );
00596             tmpsum |= unidigit[i][j];
00597         }
00598     }
00599     if (tmpsum == 0) { /* the glyph matrix is transposed */
00600         flip = 1; /* note transposed order for processing glyphs in matrix */
00601         /*
00602          * Get 5th and 6th hex digits by shifting first column header left by
00603          * 1.5 columns, thereby shifting the hex digit right after the leading
00604          * "U+nnnn" page number.
00605          */
00606         for (i = 0x08; i < 0x18; i++) {
00607             bitmap[i][7] = (bitmap[i][8] « 4) | ((bitmap[i][9] » 4) & 0xf);
00608             bitmap[i][8] = (bitmap[i][9] « 4) | ((bitmap[i][10] » 4) & 0xf);
00609         }
00610         for (i = 4; i < 6; i++) {
00611             for (j = 0; j < 4; j++) {
00612                 unidigit[i][j] =
00613                     ((unsigned)bitmap[4 * j + 8 + 1][i + 3] « 24 ) |
00614                     ((unsigned)bitmap[4 * j + 8 + 2][i + 3] « 16 ) |
00615                     ((unsigned)bitmap[4 * j + 8 + 3][i + 3] « 8 ) |
00616                     ((unsigned)bitmap[4 * j + 8 + 4][i + 3] );
00617             }
00618         }
00619     }
00620
00621     /*
00622     Now determine the Unicode plane by comparing unidigit[0..5] to
00623     the hexdigit[0x0..0xF] array.
00624     */
00625     uniplane = 0;
00626     for (i=0; i<6; i++) { /* go through one bitmap digit at a time */
00627         match = 0; /* haven't found pattern yet */
00628         for (j = 0x0; !match && j <= 0xF; j++) {
00629             if (unidigit[i][0] == hexdigit[j][0] &&
00630                 unidigit[i][1] == hexdigit[j][1] &&
00631                 unidigit[i][2] == hexdigit[j][2] &&
00632                 unidigit[i][3] == hexdigit[j][3]) { /* we found the digit */
00633                 uniplane |= j;
00634                 match = 1;
00635             }
00636         }
00637         uniplane «= 4;
00638     }
00639     uniplane »= 4;
00640 }
00641 /*
00642 Now read each glyph and print it as hex.
00643 */
00644 for (i = 0x0; i <= 0xf; i++) {
00645     for (j = 0x0; j <= 0xf; j++) {
00646         for (k = 0; k < 16; k++) {
00647             if (flip) { /* transpose glyph matrix */
00648                 thischar0[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) ];
00649                 thischar1[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 1];
00650                 thischar2[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 2];
00651                 thischar3[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 3];
00652             }
00653             else {
00654                 thischar0[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) ];
00655                 thischar1[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 1];
00656                 thischar2[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 2];

```

```

00657         thischar3[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 3];
00658     }
00659 }
00660 /*
00661     If the second half of the 16*16 character is all zeroes, this
00662     character is only 8 bits wide, so print a half-width character.
00663 */
00664 empty1 = empty2 = 1;
00665 for (k=0; (empty1 || empty2) && k < 16; k++) {
00666     if (thischar1[k] != 0) empty1 = 0;
00667     if (thischar2[k] != 0) empty2 = 0;
00668 }
00669 /*
00670     Only print this glyph if it isn't blank.
00671 */
00672 if (!empty1 || !empty2) {
00673     /*
00674         If the second half is empty, this is a half-width character.
00675         Only print the first half.
00676     */
00677     /*
00678         Original GNU Unifont format is four hexadecimal digit character
00679         code followed by a colon followed by a hex string. Add support
00680         for codes beyond the Basic Multilingual Plane.
00681
00682         Unicode ranges from U+0000 to U+10FFFF, so print either a
00683         4-digit or a 6-digit code point. Note that this software
00684         should support up to an 8-digit code point, extending beyond
00685         the normal Unicode range, but this has not been fully tested.
00686     */
00687     if (uniplane > 0xff)
00688         fprintf (outfp, "%04X%X%X:", uniplane, i, j); // 6 digit code pt.
00689     else
00690         fprintf (outfp, "%02X%X%X:", uniplane, i, j); // 4 digit code pt.
00691     for (thisrow=0; thisrow<16; thisrow++) {
00692         /*
00693             If second half is empty and we're not forcing this
00694             code point to double width, print as single width.
00695         */
00696         if (!forcewide &&
00697             empty2 && !wide[(uniplane « 8) | (i « 4) | j]) {
00698             fprintf (outfp,
00699                 "%02X",
00700                 thischar1[thisrow]);
00701         }
00702         else if (wide[(uniplane « 8) | (i « 4) | j] == 4) {
00703             /* quadruple-width; force 32nd pixel to zero */
00704             fprintf (outfp,
00705                 "%02X%02X%02X%02X",
00706                 thischar0[thisrow], thischar1[thisrow],
00707                 thischar2[thisrow], thischar3[thisrow] & 0xFE);
00708         }
00709         else { /* treat as double-width */
00710             fprintf (outfp,
00711                 "%02X%02X",
00712                 thischar1[thisrow], thischar2[thisrow]);
00713         }
00714     }
00715     fprintf (outfp, "\n");
00716 }
00717 }
00718 }
00719 exit (0);
00720 }

```

5.7.4 Variable Documentation

5.7.4.1 bits_per_pixel

```
int bits_per_pixel
```

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

5.7.4.2

```
struct { ... } bmp_header
```

Bitmap Header parameters

5.7.4.3 color_table

```
unsigned char color_table[256][4]
```

Bitmap Color [Table](#) – maximum of 256 colors in a BMP file

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

5.7.4.4 compression

```
int compression
```

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

5.7.4.5 file_size

```
int file_size
```

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

5.7.4.6 filetype

```
char filetype[2]
```

Definition at line [120](#) of file [unibmp2hex.c](#).

5.7.4.7 flip

unsigned flip =0

=1 if we're transposing glyph matrix

Definition at line [111](#) of file [unibmp2hex.c](#).

5.7.4.8 forcewide

unsigned forcewide =0

=1 to set each glyph to 16 pixels wide

Definition at line [112](#) of file [unibmp2hex.c](#).

5.7.4.9 height

int height

Definition at line [125](#) of file [unibmp2hex.c](#).

5.7.4.10 hexdigit

unsigned hexdigit[16][4]

32 bit representation of 16x8 0..F bitmap

Definition at line [107](#) of file [unibmp2hex.c](#).

5.7.4.11 image__offset

int image__offset

Definition at line [122](#) of file [unibmp2hex.c](#).

5.7.4.12 image_size

int image_size

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

5.7.4.13 important_colors

int important_colors

Definition at line 133 of file [unibmp2hex.c](#).

5.7.4.14 info_size

int info_size

Definition at line 123 of file [unibmp2hex.c](#).

5.7.4.15 ncolors

int ncolors

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

5.7.4.16 nplanes

int nplanes

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

5.7.4.17 planeset

unsigned planeset =0

=1: use plane specified with -p parameter

Definition at line 110 of file [unibmp2hex.c](#).

5.7.4.18 unidigit

unsigned unidigit[6][4]

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

Definition at line [115](#) of file [unibmp2hex.c](#).

5.7.4.19 uniplane

unsigned uniplane =0

Unicode plane number, 0..0xff ff.

Definition at line [109](#) of file [unibmp2hex.c](#).

5.7.4.20 width

int width

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

5.7.4.21 x_ppm

int x_ppm

Definition at line [130](#) of file [unibmp2hex.c](#).

5.7.4.22 y_ppm

int y_ppm

Definition at line [131](#) of file [unibmp2hex.c](#).

5.8 unibmp2hex.c

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

```

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  6 September 2021 [Paul Hardy]:
00033      - Set U+12F90..U+12FFF (Cypro-Minoan) to be double width.
00034      - Set U+1CF00..U+1CFCF (Znamenny Musical Notation) to be double width.
00035      - Set U+1AFF0..U+1AFFF (Kana Extended-B) to be double width.
00036
00037  20 June 2017 [Paul Hardy]:
00038      - Modify to allow hard-coding of quadruple-width hex glyphs.
00039      The 32nd column (rightmost column) is cleared to zero, because
00040      that column contains the vertical cell border.
00041      - Set U+9FD8..U+9FE9 (complex CJK) to be quadruple-width.
00042      - Set U+011A00..U+011A4F (Masaram Gondi, non-digits) to be wide.
00043      - Set U+011A50..U+011AAF (Soyombo) to be wide.
00044
00045  8 July 2017 [Paul Hardy]:
00046      - All CJK glyphs in the range U+4E00..u+9FFF are double width
00047      again; commented out the line that sets U+9FD8..U+9FE9 to be
00048      quadruple width.
00049
00050  6 August 2017 [Paul Hardy]:
00051      - Remove hard-coding of U+01D200..U+01D24F Ancient Greek Musical
00052      Notation to double-width; allow range to be dual-width.
00053
00054  12 August 2017 [Paul Hardy]:
00055      - Remove Miao script from list of wide scripts, so it can contain
00056      single-width glyphs.
00057
00058  26 December 2017 Paul Hardy:
00059      - Removed Tibetan from list of wide scripts, so it can contain
00060      single-width glyphs.
00061      - Added a number of scripts to be explicitly double-width in case
00062      they are redrawn.
00063      - Added Miao script back as wide, because combining glyphs are
00064      added back to font/plane01/plane01-combining.txt.
00065
00066  05 June 2018 Paul Hardy:
00067      - Made U+2329] and U+232A wide.
00068      - Added to wide settings for CJK Compatibility Forms over entire range.
00069      - Made Kayah Li script double-width.
00070      - Made U+232A (Right-pointing Angle Bracket) double-width.
00071      - Made U+01F5E7 (Three Rays Right) double-width.
00072
00073  July 2018 Paul Hardy:
00074      - Changed 2017 to 2018 in previous change entry.
00075      - Added Dogra (U+011800..U+01184F) as double width.
00076      - Added Makasar (U+011EE0..U+011EFF) as double width.

```

```

00077
00078 23 February 2019 [Paul Hardy]:
00079 - Set U+119A0..U+119FF (Nandinagari) to be wide.
00080 - Set U+1E2C0..U+1E2FF (Wancho) to be wide.
00081
00082 25 May 2019 [Paul Hardy]:
00083 - Added support for the case when the original .bmp monochrome
00084 file has been converted to a 32 bit per pixel RGB file.
00085 - Added support for bitmap images stored from either top to bottom
00086 or bottom to top.
00087 - Add DEBUG compile flag to print header information, to ease
00088 adding support for additional bitmap formats in the future.
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
00098 */
00099
00100 #include <stdio.h>
00101 #include <stdlib.h>
00102 #include <string.h>
00103
00104 #define MAXBUF 256 ///< Maximum input file line length - 1
00105
00106
00107 unsigned hexdigit[16][4]; ///< 32 bit representation of 16x8 0..F bitmap
00108
00109 unsigned uniplane=0; ///< Unicode plane number, 0..0xff ff
00110 unsigned planeset=0; ///< =1: use plane specified with -p parameter
00111 unsigned flip=0; ///< =1 if we're transposing glyph matrix
00112 unsigned forcewide=0; ///< =1 to set each glyph to 16 pixels wide
00113
00114 /** The six Unicode plane digits, from left-most (0) to right-most (5) */
00115 unsigned unidigit[6][4];
00116
00117
00118 /** Bitmap Header parameters */
00119 struct {
00120 char filetype[2];
00121 int file_size;
00122 int image_offset;
00123 int info_size;
00124 int width;
00125 int height;
00126 int nplanes;
00127 int bits_per_pixel;
00128 int compression;
00129 int image_size;
00130 int x_ppm;
00131 int y_ppm;
00132 int ncolors;
00133 int important_colors;
00134 } bmp_header;
00135
00136 /** Bitmap Color Table -- maximum of 256 colors in a BMP file */
00137 unsigned char color_table[256][4]; /* R, G, B, alpha for up to 256 colors */
00138
00139 // #define DEBUG
00140
00141 /**
00142 @brief The main function.
00143
00144 @param[in] argc The count of command line arguments.
00145 @param[in] argv Pointer to array of command line arguments.
00146 @return This program exits with status 0.
00147 */
00148 int
00149 main (int argc, char *argv[])
00150 {
00151
00152 int i, j, k; /* loop variables */
00153 unsigned char inchar; /* temporary input character */
00154 char header[MAXBUF]; /* input buffer for bitmap file header */
00155 int wbmp=0; /* =0 for Windows Bitmap (.bmp); 1 for Wireless Bitmap (.wbmp) */
00156 int fatal; /* =1 if a fatal error occurred */
00157 int match; /* =1 if we're still matching a pattern, 0 if no match */

```

```

00158 int empty1, empty2; /* =1 if bytes tested are all zeroes */
00159 unsigned char thischar1[16], thischar2[16]; /* bytes of hex char */
00160 unsigned char thischar0[16], thischar3[16]; /* bytes for quadruple-width */
00161 int thisrow; /* index to point into thischar1[] and thischar2[] */
00162 int tmpsum; /* temporary sum to see if a character is blank */
00163 unsigned this_pixel; /* color of one pixel, if > 1 bit per pixel */
00164 unsigned next_pixels; /* pending group of 8 pixels being read */
00165 unsigned color_mask = 0x00; /* to invert monochrome bitmap, set to 0xFF */
00166
00167 unsigned char bitmap[17*32][18*32/8]; /* final bitmap */
00168 /* For wide array:
00169     0 = don't force glyph to double-width;
00170     1 = force glyph to double-width;
00171     4 = force glyph to quadruple-width.
00172 */
00173 char wide[0x200000]={0x200000 * 0};
00174
00175 char *infile="", *outfile=""; /* names of input and output files */
00176 FILE *infp, *outfp; /* file pointers of input and output files */
00177
00178 if (argc > 1) {
00179     for (i = 1; i < argc; i++) {
00180         if (argv[i][0] == '-') { /* this is an option argument */
00181             switch (argv[i][1]) {
00182                 case 'i': /* name of input file */
00183                     infile = &argv[i][2];
00184                     break;
00185                 case 'o': /* name of output file */
00186                     outfile = &argv[i][2];
00187                     break;
00188                 case 'p': /* specify a Unicode plane */
00189                     sscanf (&argv[i][2], "%x", &uniplane); /* Get Unicode plane */
00190                     planeset = 1; /* Use specified range, not what's in bitmap */
00191                     break;
00192                 case 'w': /* force wide (16 pixels) for each glyph */
00193                     forcewide = 1;
00194                     break;
00195                 default: /* if unrecognized option, print list and exit */
00196                     fprintf (stderr, "\nSyntax:\n\n");
00197                     fprintf (stderr, " %s -p<Unicode_Page> ", argv[0]);
00198                     fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
00199                     fprintf (stderr, " -w specifies .wbmp output instead of ");
00200                     fprintf (stderr, "default Windows .bmp output.\n\n");
00201                     fprintf (stderr, " -p is followed by 1 to 6 ");
00202                     fprintf (stderr, "Unicode plane hex digits ");
00203                     fprintf (stderr, "(default is Page 0).\n\n");
00204                     fprintf (stderr, "\nExample:\n\n");
00205                     fprintf (stderr, " %s -p83 -iunifont.hex -ou83.bmp\n\n",
00206                             argv[0]);
00207                     exit (1);
00208             }
00209         }
00210     }
00211 }
00212 /*
00213     Make sure we can open any I/O files that were specified before
00214     doing anything else.
00215 */
00216 if (strlen (infile) > 0) {
00217     if ((infp = fopen (infile, "r")) == NULL) {
00218         fprintf (stderr, "Error: can't open %s for input.\n", infile);
00219         exit (1);
00220     }
00221 }
00222 else {
00223     infp = stdin;
00224 }
00225 if (strlen (outfile) > 0) {
00226     if ((outfp = fopen (outfile, "w")) == NULL) {
00227         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00228         exit (1);
00229     }
00230 }
00231 else {
00232     outfp = stdout;
00233 }
00234 /*
00235     Initialize selected code points for double width (16x16).
00236     Double-width is forced in cases where a glyph (usually a combining
00237     glyph) only occupies the left-hand side of a 16x16 grid, but must
00238     be rendered as double-width to appear properly with other glyphs

```

```

00239     in a given script. If additions were made to a script after
00240     Unicode 5.0, the Unicode version is given in parentheses after
00241     the script name.
00242 */
00243 for (i = 0x0700; i <= 0x074F; i++) wide[i] = 1; /* Syriac */
00244 for (i = 0x0800; i <= 0x083F; i++) wide[i] = 1; /* Samaritan (5.2) */
00245 for (i = 0x0900; i <= 0x0DFF; i++) wide[i] = 1; /* Indic */
00246 for (i = 0x1000; i <= 0x109F; i++) wide[i] = 1; /* Myanmar */
00247 for (i = 0x1100; i <= 0x11FF; i++) wide[i] = 1; /* Hangul Jamo */
00248 for (i = 0x1400; i <= 0x167F; i++) wide[i] = 1; /* Canadian Aboriginal */
00249 for (i = 0x1700; i <= 0x171F; i++) wide[i] = 1; /* Tagalog */
00250 for (i = 0x1720; i <= 0x173F; i++) wide[i] = 1; /* Hanunoo */
00251 for (i = 0x1740; i <= 0x175F; i++) wide[i] = 1; /* Buhid */
00252 for (i = 0x1760; i <= 0x177F; i++) wide[i] = 1; /* Tagbanwa */
00253 for (i = 0x1780; i <= 0x17FF; i++) wide[i] = 1; /* Khmer */
00254 for (i = 0x18B0; i <= 0x18FF; i++) wide[i] = 1; /* Ext. Can. Aboriginal */
00255 for (i = 0x1800; i <= 0x18AF; i++) wide[i] = 1; /* Mongolian */
00256 for (i = 0x1900; i <= 0x194F; i++) wide[i] = 1; /* Limbu */
00257 // for (i = 0x1980; i <= 0x19DF; i++) wide[i] = 1; /* New Tai Lue */
00258 for (i = 0x1A00; i <= 0x1A1F; i++) wide[i] = 1; /* Buginese */
00259 for (i = 0x1A20; i <= 0x1AAF; i++) wide[i] = 1; /* Tai Tham (5.2) */
00260 for (i = 0x1B00; i <= 0x1B7F; i++) wide[i] = 1; /* Balinese */
00261 for (i = 0x1B80; i <= 0x1BBF; i++) wide[i] = 1; /* Sundanese (5.1) */
00262 for (i = 0x1BC0; i <= 0x1BFF; i++) wide[i] = 1; /* Batak (6.0) */
00263 for (i = 0x1C00; i <= 0x1C4F; i++) wide[i] = 1; /* Lepcha (5.1) */
00264 for (i = 0x1CC0; i <= 0x1CCF; i++) wide[i] = 1; /* Sundanese Supplement */
00265 for (i = 0x1CD0; i <= 0x1CFF; i++) wide[i] = 1; /* Vedic Extensions (5.2) */
00266 wide[0x2329] = wide[0x232A] = 1; /* Left- & Right-pointing Angle Brackets */
00267 for (i = 0x2E80; i <= 0xA4CF; i++) wide[i] = 1; /* CJK */
00268 // for (i = 0x9FD8; i <= 0x9FE9; i++) wide[i] = 4; /* CJK quadruple-width */
00269 for (i = 0xA900; i <= 0xA92F; i++) wide[i] = 1; /* Kayah Li (5.1) */
00270 for (i = 0xA930; i <= 0xA95F; i++) wide[i] = 1; /* Rejang (5.1) */
00271 for (i = 0xA960; i <= 0xA97F; i++) wide[i] = 1; /* Hangul Jamo Extended-A */
00272 for (i = 0xA980; i <= 0xA9DF; i++) wide[i] = 1; /* Javanese (5.2) */
00273 for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham (5.1) */
00274 for (i = 0xA9E0; i <= 0xA9FF; i++) wide[i] = 1; /* Myanmar Extended-B */
00275 for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham */
00276 for (i = 0xAA60; i <= 0xAA7F; i++) wide[i] = 1; /* Myanmar Extended-A */
00277 for (i = 0xAAE0; i <= 0xA AFF; i++) wide[i] = 1; /* Meetei Mayek Ext (6.0) */
00278 for (i = 0xABC0; i <= 0xABFF; i++) wide[i] = 1; /* Meetei Mayek (5.2) */
00279 for (i = 0xAC00; i <= 0xD7AF; i++) wide[i] = 1; /* Hangul Syllables */
00280 for (i = 0xD7B0; i <= 0xD7FF; i++) wide[i] = 1; /* Hangul Jamo Extended-B */
00281 for (i = 0xF900; i <= 0xFAFF; i++) wide[i] = 1; /* CJK Compatibility */
00282 for (i = 0xFE10; i <= 0xFE1F; i++) wide[i] = 1; /* Vertical Forms */
00283 for (i = 0xFE30; i <= 0xFE60; i++) wide[i] = 1; /* CJK Compatibility Forms */
00284 for (i = 0xFFE0; i <= 0xFFE6; i++) wide[i] = 1; /* CJK Compatibility Forms */
00285
00286 wide[0x303F] = 0; /* CJK half-space fill */
00287
00288 /* Supplemental Multilingual Plane (Plane 01) */
00289 for (i = 0x010A00; i <= 0x010A5F; i++) wide[i] = 1; /* Kharoshthi */
00290 for (i = 0x011000; i <= 0x01107F; i++) wide[i] = 1; /* Brahmi */
00291 for (i = 0x011080; i <= 0x0110CF; i++) wide[i] = 1; /* Kaithi */
00292 for (i = 0x011100; i <= 0x01114F; i++) wide[i] = 1; /* Chakma */
00293 for (i = 0x011180; i <= 0x0111DF; i++) wide[i] = 1; /* Sharada */
00294 for (i = 0x011200; i <= 0x01124F; i++) wide[i] = 1; /* Khojki */
00295 for (i = 0x0112B0; i <= 0x0112FF; i++) wide[i] = 1; /* Khudawadi */
00296 for (i = 0x011300; i <= 0x01137F; i++) wide[i] = 1; /* Grantha */
00297 for (i = 0x011400; i <= 0x01147F; i++) wide[i] = 1; /* Newa */
00298 for (i = 0x011480; i <= 0x0114DF; i++) wide[i] = 1; /* Tirhuta */
00299 for (i = 0x011580; i <= 0x0115FF; i++) wide[i] = 1; /* Siddham */
00300 for (i = 0x011600; i <= 0x01165F; i++) wide[i] = 1; /* Modi */
00301 for (i = 0x011660; i <= 0x01167F; i++) wide[i] = 1; /* Mongolian Suppl. */
00302 for (i = 0x011680; i <= 0x0116CF; i++) wide[i] = 1; /* Takri */
00303 for (i = 0x011700; i <= 0x01173F; i++) wide[i] = 1; /* Ahom */
00304 for (i = 0x011800; i <= 0x01184F; i++) wide[i] = 1; /* Dogra */
00305 for (i = 0x011900; i <= 0x01195F; i++) wide[i] = 1; /* Dives Akuru */
00306 for (i = 0x0119A0; i <= 0x0119FF; i++) wide[i] = 1; /* Nandinagari */
00307 for (i = 0x011A00; i <= 0x011A4F; i++) wide[i] = 1; /* Zanabazar Square */
00308 for (i = 0x011A50; i <= 0x011AAF; i++) wide[i] = 1; /* Soyombo */
00309 for (i = 0x011B00; i <= 0x011B5F; i++) wide[i] = 1; /* Devanagari Extended-A */
00310 for (i = 0x011F00; i <= 0x011F5F; i++) wide[i] = 1; /* Kawi */
00311 for (i = 0x011C00; i <= 0x011C6F; i++) wide[i] = 1; /* Bhaiksuki */
00312 for (i = 0x011C70; i <= 0x011CBF; i++) wide[i] = 1; /* Marchen */
00313 for (i = 0x011D00; i <= 0x011D5F; i++) wide[i] = 1; /* Masaram Gondi */
00314 for (i = 0x011EE0; i <= 0x011EFF; i++) wide[i] = 1; /* Makasar */
00315 for (i = 0x012F90; i <= 0x012FFF; i++) wide[i] = 1; /* Cypro-Minoan */
00316 /* Make Bassa Vah all single width or all double width */
00317 for (i = 0x016AD0; i <= 0x016AFF; i++) wide[i] = 1; /* Bassa Vah */
00318 for (i = 0x016B00; i <= 0x016B8F; i++) wide[i] = 1; /* Pahawh Hmong */
00319 for (i = 0x016F00; i <= 0x016F9F; i++) wide[i] = 1; /* Miao */

```

```

00320 for (i = 0x016FE0; i <= 0x016FFF; i++) wide[i] = 1; /* Ideograph Sym/Punct */
00321 for (i = 0x017000; i <= 0x0187FF; i++) wide[i] = 1; /* Tangut */
00322 for (i = 0x018800; i <= 0x018AFF; i++) wide[i] = 1; /* Tangut Components */
00323 for (i = 0x01AFF0; i <= 0x01AFFF; i++) wide[i] = 1; /* Kana Extended-B */
00324 for (i = 0x01B000; i <= 0x01B0FF; i++) wide[i] = 1; /* Kana Supplement */
00325 for (i = 0x01B100; i <= 0x01B12F; i++) wide[i] = 1; /* Kana Extended-A */
00326 for (i = 0x01B170; i <= 0x01B2FF; i++) wide[i] = 1; /* Nushu */
00327 for (i = 0x01CF00; i <= 0x01CFCF; i++) wide[i] = 1; /* Znamenny Musical */
00328 for (i = 0x01D100; i <= 0x01D1FF; i++) wide[i] = 1; /* Musical Symbols */
00329 for (i = 0x01D800; i <= 0x01DAAF; i++) wide[i] = 1; /* Sutton SignWriting */
00330 for (i = 0x01E2C0; i <= 0x01E2FF; i++) wide[i] = 1; /* Wancho */
00331 for (i = 0x01E800; i <= 0x01E8DF; i++) wide[i] = 1; /* Mende Kikakui */
00332 for (i = 0x01F200; i <= 0x01F2FF; i++) wide[i] = 1; /* Encl Ideograp Suppl */
00333 wide[0x01F5E7] = 1; /* Three Rays Right */
00334
00335 /*
00336 Determine whether or not the file is a Microsoft Windows Bitmap file.
00337 If it starts with 'B', 'M', assume it's a Windows Bitmap file.
00338 Otherwise, assume it's a Wireless Bitmap file.
00339
00340 WARNING: There isn't much in the way of error checking here --
00341 if you give it a file that wasn't first created by hex2bmp.c,
00342 all bets are off.
00343 */
00344 fatal = 0; /* assume everything is okay with reading input file */
00345 if ((header[0] = fgetc (infp)) != EOF) {
00346     if ((header[1] = fgetc (infp)) != EOF) {
00347         if (header[0] == 'B' && header[1] == 'M') {
00348             wbmp = 0; /* Not a Wireless Bitmap -- it's a Windows Bitmap */
00349         }
00350         else {
00351             wbmp = 1; /* Assume it's a Wireless Bitmap */
00352         }
00353     }
00354     else
00355         fatal = 1;
00356 }
00357 else
00358     fatal = 1;
00359
00360 if (fatal) {
00361     fprintf (stderr, "Fatal error; end of input file.\n\n");
00362     exit (1);
00363 }
00364 /*
00365 If this is a Wireless Bitmap (.wbmp) format file,
00366 skip the header and point to the start of the bitmap itself.
00367 */
00368 if (wbmp) {
00369     for (i=2; i<6; i++)
00370         header[i] = fgetc (infp);
00371     /*
00372     Now read the bitmap.
00373     */
00374     for (i=0; i < 32*17; i++) {
00375         for (j=0; j < 32*18/8; j++) {
00376             inchar = fgetc (infp);
00377             bitmap[i][j] = ~inchar; /* invert bits for proper color */
00378         }
00379     }
00380 }
00381 /*
00382 Otherwise, treat this as a Windows Bitmap file, because we checked
00383 that it began with "BM". Save the header contents for future use.
00384 Expect a 14 byte standard BITMAPFILEHEADER format header followed
00385 by a 40 byte standard BITMAPINFOHEADER Device Independent Bitmap
00386 header, with data stored in little-endian format.
00387 */
00388 else {
00389     for (i = 2; i < 54; i++)
00390         header[i] = fgetc (infp);
00391
00392     bmp_header.filetype[0] = 'B';
00393     bmp_header.filetype[1] = 'M';
00394
00395     bmp_header.file_size =
00396         (header[2] & 0xFF) | ((header[3] & 0xFF) << 8) |
00397         ((header[4] & 0xFF) << 16) | ((header[5] & 0xFF) << 24);
00398
00399     /* header bytes 6..9 are reserved */
00400

```

```

00401     bmp_header.image_offset =
00402         (header[10] & 0xFF) | ((header[11] & 0xFF) << 8) |
00403         ((header[12] & 0xFF) << 16) | ((header[13] & 0xFF) << 24);
00404
00405     bmp_header.info_size =
00406         (header[14] & 0xFF) | ((header[15] & 0xFF) << 8) |
00407         ((header[16] & 0xFF) << 16) | ((header[17] & 0xFF) << 24);
00408
00409     bmp_header.width =
00410         (header[18] & 0xFF) | ((header[19] & 0xFF) << 8) |
00411         ((header[20] & 0xFF) << 16) | ((header[21] & 0xFF) << 24);
00412
00413     bmp_header.height =
00414         (header[22] & 0xFF) | ((header[23] & 0xFF) << 8) |
00415         ((header[24] & 0xFF) << 16) | ((header[25] & 0xFF) << 24);
00416
00417     bmp_header.nplanes =
00418         (header[26] & 0xFF) | ((header[27] & 0xFF) << 8);
00419
00420     bmp_header.bits_per_pixel =
00421         (header[28] & 0xFF) | ((header[29] & 0xFF) << 8);
00422
00423     bmp_header.compression =
00424         (header[30] & 0xFF) | ((header[31] & 0xFF) << 8) |
00425         ((header[32] & 0xFF) << 16) | ((header[33] & 0xFF) << 24);
00426
00427     bmp_header.image_size =
00428         (header[34] & 0xFF) | ((header[35] & 0xFF) << 8) |
00429         ((header[36] & 0xFF) << 16) | ((header[37] & 0xFF) << 24);
00430
00431     bmp_header.x_ppm =
00432         (header[38] & 0xFF) | ((header[39] & 0xFF) << 8) |
00433         ((header[40] & 0xFF) << 16) | ((header[41] & 0xFF) << 24);
00434
00435     bmp_header.y_ppm =
00436         (header[42] & 0xFF) | ((header[43] & 0xFF) << 8) |
00437         ((header[44] & 0xFF) << 16) | ((header[45] & 0xFF) << 24);
00438
00439     bmp_header.ncolors =
00440         (header[46] & 0xFF) | ((header[47] & 0xFF) << 8) |
00441         ((header[48] & 0xFF) << 16) | ((header[49] & 0xFF) << 24);
00442
00443     bmp_header.important_colors =
00444         (header[50] & 0xFF) | ((header[51] & 0xFF) << 8) |
00445         ((header[52] & 0xFF) << 16) | ((header[53] & 0xFF) << 24);
00446
00447     if (bmp_header.ncolors == 0)
00448         bmp_header.ncolors = 1 << bmp_header.bits_per_pixel;
00449
00450     /* If a Color Table exists, read it */
00451     if (bmp_header.ncolors > 0 && bmp_header.bits_per_pixel <= 8) {
00452         for (i = 0; i < bmp_header.ncolors; i++) {
00453             color_table[i][0] = fgetc (infp); /* Red */
00454             color_table[i][1] = fgetc (infp); /* Green */
00455             color_table[i][2] = fgetc (infp); /* Blue */
00456             color_table[i][3] = fgetc (infp); /* Alpha */
00457         }
00458         /*
00459          Determine from the first color table entry whether we
00460          are inverting the resulting bitmap image.
00461          */
00462         if ( ( color_table[0][0] + color_table[0][1] + color_table[0][2])
00463              < (3 * 128) ) {
00464             color_mask = 0xFF;
00465         }
00466     }
00467
00468 #ifndef DEBUG
00469
00470     /*
00471      Print header info for possibly adding support for
00472      additional file formats in the future, to determine
00473      how the bitmap is encoded.
00474      */
00475     fprintf (stderr, "Filetype: '%c%c'\n",
00476             bmp_header.filetype[0], bmp_header.filetype[1]);
00477     fprintf (stderr, "File Size: %d\n", bmp_header.file_size);
00478     fprintf (stderr, "Image Offset: %d\n", bmp_header.image_offset);
00479     fprintf (stderr, "Info Header Size: %d\n", bmp_header.info_size);
00480     fprintf (stderr, "Image Width: %d\n", bmp_header.width);
00481     fprintf (stderr, "Image Height: %d\n", bmp_header.height);

```

```

00482     fprintf(stderr, "Number of Planes: %d\n", bmp_header.nplanes);
00483     fprintf(stderr, "Bits per Pixel: %d\n", bmp_header.bits_per_pixel);
00484     fprintf(stderr, "Compression Method: %d\n", bmp_header.compression);
00485     fprintf(stderr, "Image Size: %d\n", bmp_header.image_size);
00486     fprintf(stderr, "X Pixels per Meter: %d\n", bmp_header.x_ppm);
00487     fprintf(stderr, "Y Pixels per Meter: %d\n", bmp_header.y_ppm);
00488     fprintf(stderr, "Number of Colors: %d\n", bmp_header.ncolors);
00489     fprintf(stderr, "Important Colors: %d\n", bmp_header.important_colors);
00490
00491 #endif
00492
00493 /*
00494  * Now read the bitmap.
00495  */
00496 for (i = 32*17-1; i >= 0; i--) {
00497     for (j=0; j < 32*18/8; j++) {
00498         next_pixels = 0x00; /* initialize next group of 8 pixels */
00499         /* Read a monochrome image -- the original case */
00500         if (bmp_header.bits_per_pixel == 1) {
00501             next_pixels = fgetc (infp);
00502         }
00503         /* Read a 32 bit per pixel RGB image; convert to monochrome */
00504         else if (bmp_header.bits_per_pixel == 24 ||
00505                 bmp_header.bits_per_pixel == 32) {
00506             next_pixels = 0;
00507             for (k = 0; k < 8; k++) { /* get next 8 pixels */
00508                 this_pixel = (fgetc (infp) & 0xFF) +
00509                     (fgetc (infp) & 0xFF) +
00510                     (fgetc (infp) & 0xFF);
00511
00512                 if (bmp_header.bits_per_pixel == 32) {
00513                     (void) fgetc (infp); /* ignore alpha value */
00514                 }
00515
00516                 /* convert RGB color space to monochrome */
00517                 if (this_pixel >= (128 * 3))
00518                     this_pixel = 0;
00519                 else
00520                     this_pixel = 1;
00521
00522                 /* shift next pixel color into place for 8 pixels total */
00523                 next_pixels = (next_pixels << 1) | this_pixel;
00524             }
00525         }
00526         if (bmp_header.height < 0) { /* Bitmap drawn top to bottom */
00527             bitmap [(32*17-1) - i] [j] = next_pixels;
00528         }
00529         else { /* Bitmap drawn bottom to top */
00530             bitmap [i] [j] = next_pixels;
00531         }
00532     }
00533 }
00534
00535 /*
00536  * If any bits are set in color_mask, apply it to
00537  * entire bitmap to invert black <--> white.
00538  */
00539 if (color_mask != 0x00) {
00540     for (i = 32*17-1; i >= 0; i--) {
00541         for (j=0; j < 32*18/8; j++) {
00542             bitmap [i] [j] ^= color_mask;
00543         }
00544     }
00545 }
00546 }
00547
00548 /*
00549  * We've read the entire file. Now close the input file pointer.
00550  */
00551 fclose (infp);
00552
00553 /*
00554  * We now have the header portion in the header[] array,
00555  * and have the bitmap portion from top-to-bottom in the bitmap[] array.
00556  */
00557 /*
00558  * If no Unicode range (U+nnnnnn00 through U+nnnnnnFF) was specified
00559  * with a -p parameter, determine the range from the digits in the
00560  * bitmap itself.
00561
00562  * Store bitmaps for the hex digit patterns that this file uses.

```



```

00563 */
00564 if (!planeset) { /* If Unicode range not specified with -p parameter */
00565     for (i = 0x0; i <= 0xF; i++) { /* hex digit pattern we're storing */
00566         for (j = 0; j < 4; j++) {
00567             hexdigit[i][j] =
00568                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 ][6] << 24 ) |
00569                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 1][6] << 16 ) |
00570                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 2][6] << 8 ) |
00571                 ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 3][6] );
00572         }
00573     }
00574     /*
00575     Read the Unicode plane digits into arrays for comparison, to
00576     determine the upper four hex digits of the glyph addresses.
00577     */
00578     for (i = 0; i < 4; i++) {
00579         for (j = 0; j < 4; j++) {
00580             unidigit[i][j] =
00581                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 1][i + 3] << 24 ) |
00582                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 2][i + 3] << 16 ) |
00583                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 3][i + 3] << 8 ) |
00584                 ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 4][i + 3] );
00585         }
00586     }
00587
00588     tmpsum = 0;
00589     for (i = 4; i < 6; i++) {
00590         for (j = 0; j < 4; j++) {
00591             unidigit[i][j] =
00592                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 ][i] << 24 ) |
00593                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 1][i] << 16 ) |
00594                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 2][i] << 8 ) |
00595                 ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 3][i] );
00596             tmpsum |= unidigit[i][j];
00597         }
00598     }
00599     if (tmpsum == 0) { /* the glyph matrix is transposed */
00600         flip = 1; /* note transposed order for processing glyphs in matrix */
00601         /*
00602         Get 5th and 6th hex digits by shifting first column header left by
00603         1.5 columns, thereby shifting the hex digit right after the leading
00604         "U+nnnn" page number.
00605         */
00606         for (i = 0x08; i < 0x18; i++) {
00607             bitmap[i][7] = (bitmap[i][8] << 4) | ((bitmap[i][9] >> 4) & 0xf);
00608             bitmap[i][8] = (bitmap[i][9] << 4) | ((bitmap[i][10] >> 4) & 0xf);
00609         }
00610         for (i = 4; i < 6; i++) {
00611             for (j = 0; j < 4; j++) {
00612                 unidigit[i][j] =
00613                     ((unsigned)bitmap[4 * j + 8 + 1][i + 3] << 24 ) |
00614                     ((unsigned)bitmap[4 * j + 8 + 2][i + 3] << 16 ) |
00615                     ((unsigned)bitmap[4 * j + 8 + 3][i + 3] << 8 ) |
00616                     ((unsigned)bitmap[4 * j + 8 + 4][i + 3] );
00617             }
00618         }
00619     }
00620
00621     /*
00622     Now determine the Unicode plane by comparing unidigit[0..5] to
00623     the hexdigit[0x0..0xF] array.
00624     */
00625     uniplane = 0;
00626     for (i=0; i<6; i++) { /* go through one bitmap digit at a time */
00627         match = 0; /* haven't found pattern yet */
00628         for (j = 0x0; !match && j <= 0xF; j++) {
00629             if (unidigit[i][0] == hexdigit[j][0] &&
00630                 unidigit[i][1] == hexdigit[j][1] &&
00631                 unidigit[i][2] == hexdigit[j][2] &&
00632                 unidigit[i][3] == hexdigit[j][3]) { /* we found the digit */
00633                 uniplane |= j;
00634                 match = 1;
00635             }
00636         }
00637         uniplane <<= 4;
00638     }
00639     uniplane >>= 4;
00640 }
00641 /*
00642 Now read each glyph and print it as hex.
00643 */

```



```

00644     for (i = 0x0; i <= 0xf; i++) {
00645         for (j = 0x0; j <= 0xf; j++) {
00646             for (k = 0; k < 16; k++) {
00647                 if (flip) { /* transpose glyph matrix */
00648                     thischar0[k] = bitmap[32*(j+1) + k + 7][4 * (i+2)   ];
00649                     thischar1[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 1];
00650                     thischar2[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 2];
00651                     thischar3[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 3];
00652                 }
00653                 else {
00654                     thischar0[k] = bitmap[32*(i+1) + k + 7][4 * (j+2)   ];
00655                     thischar1[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 1];
00656                     thischar2[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 2];
00657                     thischar3[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 3];
00658                 }
00659             }
00660         } /*
00661         If the second half of the 16*16 character is all zeroes, this
00662         character is only 8 bits wide, so print a half-width character.
00663     */
00664     empty1 = empty2 = 1;
00665     for (k=0; (empty1 || empty2) && k < 16; k++) {
00666         if (thischar1[k] != 0) empty1 = 0;
00667         if (thischar2[k] != 0) empty2 = 0;
00668     }
00669     /*
00670     Only print this glyph if it isn't blank.
00671     */
00672     if (!empty1 || !empty2) {
00673         /*
00674         If the second half is empty, this is a half-width character.
00675         Only print the first half.
00676         */
00677         /*
00678         Original GNU Unifont format is four hexadecimal digit character
00679         code followed by a colon followed by a hex string. Add support
00680         for codes beyond the Basic Multilingual Plane.
00681
00682         Unicode ranges from U+0000 to U+10FFFF, so print either a
00683         4-digit or a 6-digit code point. Note that this software
00684         should support up to an 8-digit code point, extending beyond
00685         the normal Unicode range, but this has not been fully tested.
00686         */
00687         if (uniplane > 0xff)
00688             fprintf (outfp, "%04X%X%X:X:", uniplane, i, j); // 6 digit code pt.
00689         else
00690             fprintf (outfp, "%02X%X%X:X:", uniplane, i, j); // 4 digit code pt.
00691         for (thisrow=0; thisrow<16; thisrow++) {
00692             /*
00693             If second half is empty and we're not forcing this
00694             code point to double width, print as single width.
00695             */
00696             if (!forcewide &&
00697                 empty2 && !wide[(uniplane « 8) | (i « 4) | j]) {
00698                 fprintf (outfp,
00699                     "%02X",
00700                     thischar1[thisrow]);
00701             }
00702             else if (wide[(uniplane « 8) | (i « 4) | j] == 4) {
00703                 /* quadruple-width; force 32nd pixel to zero */
00704                 fprintf (outfp,
00705                     "%02X%02X%02X%02X",
00706                     thischar0[thisrow], thischar1[thisrow],
00707                     thischar2[thisrow], thischar3[thisrow] & 0xFE);
00708             }
00709             else { /* treat as double-width */
00710                 fprintf (outfp,
00711                     "%02X%02X",
00712                     thischar1[thisrow], thischar2[thisrow]);
00713             }
00714         }
00715         fprintf (outfp, "\n");
00716     }
00717 }
00718 }
00719 exit (0);
00720 }

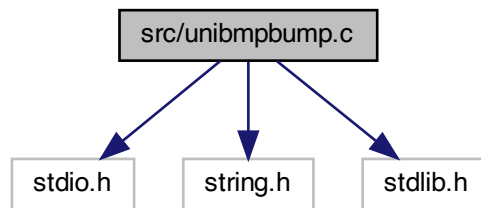
```

5.9 src/unibmpbump.c File Reference

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

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

Include dependency graph for unibmpbump.c:



Macros

- `#define` `VERSION` "1.0"
Version of this program.
- `#define` `MAX_COMPRESSION_METHOD` 13
Maximum supported compression method.

Functions

- `int` `main` (`int argc`, `char *argv[]`)
The main function.
- `unsigned` `get_bytes` (`FILE *infp`, `int nbytes`)
Get from 1 to 4 bytes, inclusive, from input file.
- `void` `regrid` (`unsigned *image_bytes`)
After reading in the image, shift it.

5.9.1 Detailed Description

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

Author

Paul Hardy, unifoundry <at> unifoundry.com

Copyright

Copyright (C) 2019 Paul Hardy

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

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

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

5.9.2 Macro Definition Documentation

5.9.2.1 MAX_COMPRESSION_METHOD

```
#define MAX_COMPRESSION_METHOD 13
```

Maximum supported compression method.

Definition at line 40 of file [unibmpbump.c](#).

5.9.2.2 VERSION

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

Version of this program.

Definition at line 38 of file [unibmpbump.c](#).

5.9.3 Function Documentation

5.9.3.1 get_bytes()

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

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

Parameters

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

Returns

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

Definition at line 487 of file [unibmpbump.c](#).

```

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

5.9.3.2 main()

```

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

The main function.

Parameters

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

Returns

This program exits with status EXIT_SUCCESS.

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

```

00050     {
00051
00052     /*
00053     /* Values preserved from file header (first 14 bytes).
00054     /*
00055     char file_format[3]; /* "BM" for original Windows format */
00056     unsigned filesize; /* size of file in bytes */
00057     unsigned char rsvd_hdr[4]; /* 4 reserved bytes */
00058     unsigned image_start; /* byte offset of image in file */
00059
00060     /*
00061     /* Values preserved from Device Independent Bitmap (DIB) Header.
00062
00063     The DIB fields below are in the standard 40-byte header. Version
00064     4 and version 5 headers have more information, mainly for color
00065     information. That is skipped over, because a valid glyph image
00066     is just monochrome.
00067     /*
00068     int dib_length; /* in bytes, for parsing by header version */
00069     int image_width = 0; /* Signed image width */
00070     int image_height = 0; /* Signed image height */
00071     int num_planes = 0; /* number of planes; must be 1 */
00072     int bits_per_pixel = 0; /* for palletized color maps (< 2^16 colors) */
00073     /*
00074     The following fields are not in the original spec, so initialize
00075     them to 0 so we can correctly parse an original file format.
00076     /*
00077     int compression_method=0; /* 0 --> uncompressed RGB/monochrome */
00078     int image_size = 0; /* 0 is a valid size if no compression */
00079     int hres = 0; /* image horizontal resolution */
00080     int vres = 0; /* image vertical resolution */
00081     int num_colors = 0; /* Number of colors for 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 /* 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 /* Read Reserved bytes */
00220 rsvd_hdr[0] = get_bytes (infp, 1);
00221 rsvd_hdr[1] = get_bytes (infp, 1);
00222 rsvd_hdr[2] = get_bytes (infp, 1);
00223 rsvd_hdr[3] = get_bytes (infp, 1);
00224
00225 /* Read Image Offset Address within file */
00226 image_start = get_bytes (infp, 4);
00227
00228 /*
00229  See if this looks like a valid image file based on
00230  the file header first two bytes.
00231 */
00232 if (strncmp (file_format, "BM", 2) != 0) {
00233     fprintf (stderr, "\nInvalid file format: not file type \"BM\".\n\n");
00234     exit (EXIT_FAILURE);
00235 }
00236
00237 if (verbose) {
00238     fprintf (stderr, "\nFile Header:\n");
00239     fprintf (stderr, "  File Type:  \"%s\"\n", file_format);
00240     fprintf (stderr, "  File Size:  %d bytes\n", filesize);
00241     fprintf (stderr, "  Reserved:  ");
00242     for (i = 0; i < 4; i++) fprintf (stderr, " 0x%02X", rsvd_hdr[i]);
00243     fputc ('\n', stderr);
00244     fprintf (stderr, "  Image Start: %d. = 0x%02X = 0x%05o\n\n",
00245             image_start, image_start, image_start);
00246 } /* if (verbose) */
00247
00248 /*
00249  Device Independent Bitmap (DIB) Header: bitmap information header
00250  ("BM" format file DIB Header is 12 bytes long).
00251 */
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 /*
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
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 }

```

5.9.3.3 regrid()

```
void regrid (
    unsigned * image_bytes )
```

After reading in the image, shift it.

This function adjusts the input image from an original PNG file to match [unihex2bmp.c](#) format.

Parameters

in,out	image_bytes	The pixels in an image.
--------	-------------	-------------------------

Definition at line 514 of file [unibmpbump.c](#).

```

00514     {
00515     int i, j, k; /* loop variables */
00516     int offset;
00517     unsigned glyph_row; /* one grid row of 32 pixels */
00518     unsigned last_pixel; /* last pixel in a byte, to preserve */
00519
00520     /* To insert "00" after "U+" at top of image */
00521     char zero_pattern[16] = {
00522         0x00, 0x00, 0x00, 0x00, 0x18, 0x24, 0x42, 0x42,
00523         0x42, 0x42, 0x42, 0x42, 0x24, 0x18, 0x00, 0x00
00524     };
00525
00526     /* This is the horizontal grid pattern on glyph boundaries */
00527     unsigned hgrid[72] = {
00528         /* 0 */ 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xfe,
00529         /* 8 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00530         /* 16 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00531         /* 24 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00532         /* 32 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00533         /* 40 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00534         /* 48 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00535         /* 56 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00536         /* 64 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00
00537     };
00538
00539
00540     /*
00541     First move "U+" left and insert "00" after it.
00542     */
00543     j = 15; /* rows are written bottom to top, so we'll decrement j */
00544     for (i = 543 - 8; i > 544 - 24; i--) {
00545         offset = 72 * i;
00546         image_bytes[offset + 0] = image_bytes[offset + 2];
00547         image_bytes[offset + 1] = image_bytes[offset + 3];
00548         image_bytes[offset + 2] = image_bytes[offset + 4];
00549         image_bytes[offset + 3] = image_bytes[offset + 4] =
00550             ~zero_pattern[15 - j--] & 0xFF;
00551     }
00552
00553     /*
00554     Now move glyph bitmaps to the right by 8 pixels.
00555     */
00556     for (i = 0; i < 16; i++) { /* for each glyph row */
00557         for (j = 0; j < 16; j++) { /* for each glyph column */
00558             /* set offset to lower left-hand byte of next glyph */
00559             offset = (32 * 72 * i) + (9 * 72) + (4 * j) + 8;
00560             for (k = 0; k < 16; k++) { /* for each glyph row */
00561                 glyph_row = (image_bytes[offset + 0] << 24) |
00562                     (image_bytes[offset + 1] << 16) |
00563                     (image_bytes[offset + 2] << 8) |
00564                     (image_bytes[offset + 3]);
00565                 last_pixel = glyph_row & 1; /* preserve border */
00566                 glyph_row >>= 4;
00567                 glyph_row &= 0x0FFFFFFFE;
00568                 /* Set left 4 pixels to white and preserve last pixel */
00569                 glyph_row |= 0xF0000000 | last_pixel;
00570                 image_bytes[offset + 3] = glyph_row & 0xFF;
00571                 glyph_row >>= 8;
00572                 image_bytes[offset + 2] = glyph_row & 0xFF;
00573                 glyph_row >>= 8;
00574                 image_bytes[offset + 1] = glyph_row & 0xFF;
00575                 glyph_row >>= 8;
00576                 image_bytes[offset + 0] = glyph_row & 0xFF;
00577                 offset += 72; /* move up to next row in current glyph */
00578             }
00579         }
00580     }
00581

```

```

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 }

```

5.10 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     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 /* Read Reserved bytes */
00220 rsvd_hdr[0] = get_bytes (infp, 1);
00221 rsvd_hdr[1] = get_bytes (infp, 1);
00222 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 /*
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

```



```

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

```

```

00548     image_bytes[offset + 2] = image_bytes[offset + 4];
00549     image_bytes[offset + 3] = image_bytes[offset + 4] =
00550         ~zero_pattern[15 - j--] & 0xFF;
00551 }
00552
00553 /*
00554  * Now move glyph bitmaps to the right by 8 pixels.
00555  */
00556 for (i = 0; i < 16; i++) { /* for each glyph row */
00557     for (j = 0; j < 16; j++) { /* for each glyph column */
00558         /* set offset to lower left-hand byte of next glyph */
00559         offset = (32 * 72 * i) + (9 * 72) + (4 * j) + 8;
00560         for (k = 0; k < 16; k++) { /* for each glyph row */
00561             glyph_row = (image_bytes[offset + 0] << 24) |
00562                 (image_bytes[offset + 1] << 16) |
00563                 (image_bytes[offset + 2] << 8) |
00564                 (image_bytes[offset + 3]);
00565             last_pixel = glyph_row & 1; /* preserve border */
00566             glyph_row >>= 4;
00567             glyph_row &= 0xFFFFF0;
00568             /* Set left 4 pixels to white and preserve last pixel */
00569             glyph_row |= 0xF0000000 | last_pixel;
00570             image_bytes[offset + 3] = glyph_row & 0xFF;
00571             glyph_row >>= 8;
00572             image_bytes[offset + 2] = glyph_row & 0xFF;
00573             glyph_row >>= 8;
00574             image_bytes[offset + 1] = glyph_row & 0xFF;
00575             glyph_row >>= 8;
00576             image_bytes[offset + 0] = glyph_row & 0xFF;
00577             offset += 72; /* move up to next row in current glyph */
00578         }
00579     }
00580 }
00581
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 }

```

5.11 src/unicoverage.c File Reference

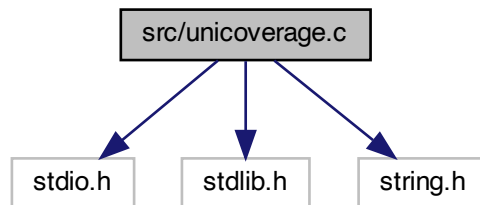
unicoverage - Show the coverage of Unicode plane scripts for a GNU Unifont hex glyph file

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

Include dependency graph for unicoverage.c:



Macros

- `#define MAXBUF 256`
Maximum input line length - 1.

Functions

- `int main (int argc, char *argv[])`
The main function.
- `int nextrange (FILE *coveragefp, int *cstart, int *cend, char *coverstring)`
Get next Unicode range.
- `void print_subtotal (FILE *outfp, int print_n, int nglyphs, int cstart, int cend, char *coverstring)`
Print the subtotal for one Unicode script range.

5.11.1 Detailed Description

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

Author

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

Copyright

Copyright (C) 2008, 2013 Paul Hardy

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

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

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

5.11.2 Macro Definition Documentation

5.11.2.1 MAXBUF

```
#define MAXBUF 256
```

Maximum input line length - 1.

Definition at line [57](#) of file [unicoverage.c](#).

5.11.3 Function Documentation

5.11.3.1 main()

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

The main function.

Parameters

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

Returns

This program exits with status 0.

Definition at line 68 of file `unicoverage.c`.

```

00069 {
00070
00071     int    print_n=0;        /* print # of glyphs, not percentage */
00072     unsigned i;              /* loop variable */
00073     unsigned slen;           /* string length of coverage file line */
00074     char    inbuf[256];      /* input buffer */
00075     unsigned thischar;       /* the current character */
00076
00077     char *infile="", *outfile=""; /* names of input and output files */
00078     FILE *infp, *outfp;      /* file pointers of input and output files */
00079     FILE *coveragefp;        /* file pointer to coverage.dat file */
00080     int cstart, cend;        /* current coverage start and end code points */
00081     char coverstring[MAXBUF]; /* description of current coverage range */
00082     int nglyphs;             /* number of glyphs in this section */
00083     int nextrange();         /* to get next range & name of Unicode glyphs */
00084
00085     void print_subtotal (FILE *outfp, int print_n, int nglyphs,
00086                         int cstart, int cend, char *coverstring);
00087
00088     if ((coveragefp = fopen ("coverage.dat", "r")) == NULL) {
00089         fprintf (stderr, "\nError: data file \"coverage.dat\" not found.\n\n");
00090         exit (0);
00091     }
00092
00093     if (argc > 1) {
00094         for (i = 1; i < argc; i++) {
00095             if (argv[i][0] == '-') { /* this is an option argument */
00096                 switch (argv[i][1]) {
00097                     case 'i': /* name of input file */
00098                         infile = &argv[i][2];
00099                         break;
00100                     case 'n': /* print number of glyphs instead of percentage */
00101                         print_n = 1;
00102                     case 'o': /* name of output file */
00103                         outfile = &argv[i][2];
00104                         break;
00105                     default: /* if unrecognized option, print list and exit */
00106                         fprintf (stderr, "\nSyntax:\n\n");
00107                         fprintf (stderr, " %s -p<Unicode_Page> ", argv[0]);
00108                         fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
00109                         exit (1);
00110                 }
00111             }

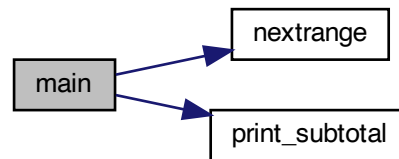
```

```

00112     }
00113 }
00114 /*
00115  Make sure we can open any I/O files that were specified before
00116  doing anything else.
00117 */
00118 if (strlen (infile) > 0) {
00119     if ((infp = fopen (infile, "r")) == NULL) {
00120         fprintf (stderr, "Error: can't open %s for input.\n", infile);
00121         exit (1);
00122     }
00123 }
00124 else {
00125     infp = stdin;
00126 }
00127 if (strlen (outfile) > 0) {
00128     if ((outfp = fopen (outfile, "w")) == NULL) {
00129         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00130         exit (1);
00131     }
00132 }
00133 else {
00134     outfp = stdout;
00135 }
00136 /*
00137  Print header row.
00138 */
00139 if (print_n) {
00140     fprintf (outfp, "# Glyphs      Range      Script\n");
00141     fprintf (outfp, "-----      ----      ----\n");
00142 }
00143 else {
00144     fprintf (outfp, "Covered      Range      Script\n");
00145     fprintf (outfp, "-----      ----      ----\n");
00146 }
00147 }
00148 slen = nextrange (coveragefp, &cstart, &cend, coverstring);
00149 nglyphs = 0;
00150
00151 /*
00152  Read in the glyphs in the file
00153 */
00154 while (slen != 0 && fgets (inbuf, MAXBUF-1, infp) != NULL) {
00155     sscanf (inbuf, "%x", &thischar);
00156
00157     /* Read a character beyond end of current script. */
00158     while (cend < thischar && slen != 0) {
00159         print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);
00160
00161         /* start new range total */
00162         slen = nextrange (coveragefp, &cstart, &cend, coverstring);
00163         nglyphs = 0;
00164     }
00165     nglyphs++;
00166 }
00167 }
00168
00169 print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);
00170
00171 exit (0);
00172 }

```

Here is the call graph for this function:



5.11.3.2 nextrange()

```

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

Get next Unicode range.

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

Parameters

in	coveragefp	File pointer to Uni-code script range data file.
in	cstart	Starting code point in current Uni-code script range.

Parameters

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

Returns

Length of the last string read, or 0 for end of file.

Definition at line 187 of file `unicoverage.c`.

```

00190 {
00191     int i;
00192     static char inbuf[MAXBUF];
00193     int retval; /* the return value */
00194
00195     retval = 0;
00196
00197     do {
00198         if (fgets (inbuf, MAXBUF-1, coveragefp) != NULL) {
00199             retval = strlen (inbuf);
00200             if ((inbuf[0] >= '0' && inbuf[0] <= '9') ||
00201                 (inbuf[0] >= 'A' && inbuf[0] <= 'F') ||
00202                 (inbuf[0] >= 'a' && inbuf[0] <= 'f')) {
00203                 sscanf (inbuf, "%x-%x", cstart, cend);
00204                 i = 0;
00205                 while (inbuf[i] != ' ') i++; /* find first blank */
00206                 while (inbuf[i] == ' ') i++; /* find next non-blank */
00207                 strncpy (coverstring, &inbuf[i], MAXBUF);
00208             }
00209             else retval = 0;
00210         }
00211         else retval = 0;
00212     } while (retval == 0 && !feof (coveragefp));
00213
00214     return (retval);
00215 }

```

Here is the caller graph for this function:



5.11.3.3 print_subtotal()

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

Print the subtotal for one Unicode script range.

Parameters

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

Parameters

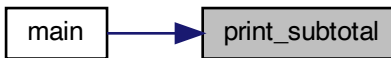
in	coverstring	Character string of "<cstart>- <cend>".
----	-------------	---

Definition at line 228 of file [unicoverage.c](#).

```

00229                                     {
00230
00231  /* print old range total */
00232  if (print_n) { /* Print number of glyphs, not percentage */
00233      fprintf (outfp, " %6d ", nglyphs);
00234  }
00235  else {
00236      fprintf (outfp, " %5.1f%%", 100.0*nglyphs/(1+cend-cstart));
00237  }
00238
00239  if (cend < 0x10000)
00240      fprintf (outfp, " U+%04X..U+%04X  %s",
00241              cstart, cend, coverstring);
00242  else
00243      fprintf (outfp, " U+%05X..U+%05X  %s",
00244              cstart, cend, coverstring);
00245
00246  return;
00247 }
```

Here is the caller graph for this function:



5.12 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 strlcpy function call to strlcpy
00040 for better error handling.
00041
00042 31 May 2019: [Paul Hardy] replaced strlcpy 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
00052 #include <stdio.h>
00053 #include <stdlib.h>
00054 #include <string.h>
00055
00056 #define MAXBUF 256 ///< Maximum input line length - 1
00057
00058 /**
00059 @brief The main function.
00060
00061 @param[in] argc The count of command line arguments.
00062 @param[in] argv Pointer to array of command line arguments.
00063 @return This program exits with status 0.
00064 */
00065 int
00066 main (int argc, char *argv[])
00067 {
00068     int print_n=0; /* print # of glyphs, not percentage */
00069     unsigned i; /* loop variable */
00070     unsigned slen; /* string length of coverage file line */
00071     char inbuf[256]; /* input buffer */
00072     unsigned thischar; /* the current character */
00073
00074     char *infile="", *outfile=""; /* names of input and output files */
00075     FILE *infp, *outfp; /* file pointers of input and output files */
00076     FILE *coveragefp; /* file pointer to coverage.dat file */
00077     int cstart, cend; /* current coverage start and end code points */
00078     char coverstring[MAXBUF]; /* description of current coverage range */
00079     int nglyphs; /* number of glyphs in this section */
00080     int nextrange(); /* to get next range & name of Unicode glyphs */
00081
00082     void print_subtotal (FILE *outfp, int print_n, int nglyphs,
00083                         int cstart, int cend, char *coverstring);
00084
00085     if ((coveragefp = fopen ("coverage.dat", "r")) == NULL) {
00086         fprintf (stderr, "\nError: data file \"coverage.dat\" not found.\n\n");
00087         exit (0);
00088     }
00089
00090     if (argc > 1) {
00091         for (i = 1; i < argc; i++) {
00092             if (argv[i][0] == '-') { /* this is an option argument */
00093                 switch (argv[i][1]) {
00094                     case 'i': /* name of input file */
00095                         infile = &argv[i][2];
00096                         break;
00097                     case 'n': /* print number of glyphs instead of percentage */

```

```

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

```

```

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

```

5.13 src/unidup.c File Reference

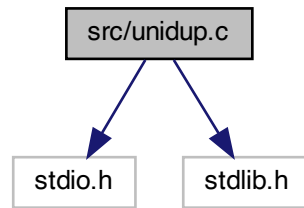
unidup - Check for duplicate code points in sorted unifont.hex file

```

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

```

Include dependency graph for unidup.c:



Macros

- `#define` [MAXBUF](#) 256
Maximum input line length - 1.

Functions

- `int` [main](#) (`int argc`, `char **argv`)
The main function.

5.13.1 Detailed Description

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

Author

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

Copyright

Copyright (C) 2007, 2008, 2013 Paul Hardy

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

Synopsis: `unidup < unifont_file.hex`

[Hopefully there won't be any output!]

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

5.13.2 Macro Definition Documentation

5.13.2.1 MAXBUF

```
#define MAXBUF 256
```

Maximum input line length - 1.

Definition at line [37](#) of file [unidup.c](#).

5.13.3 Function Documentation

5.13.3.1 main()

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

The main function.

Parameters

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

Returns

This program exits with status 0.

Definition at line 48 of file [unidup.c](#).

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

5.14 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 #include <stdio.h>
00035 #include <stdlib.h>
00036
00037 #define MAXBUF 256 ///< Maximum input line length - 1
00038
00039
00040 /**
00041     @brief The main function.
00042
00043     @param[in] argc The count of command line arguments.
```



```

00044  @param[in] argv Pointer to array of command line arguments.
00045  @return This program exits with status 0.
00046  */
00047  int
00048  main (int argc, char **argv)
00049  {
00050
00051      int ix, iy;
00052      char inbuf[MAXBUF];
00053      char *infile; /* the input file name */
00054      FILE *infilep; /* file pointer to input file */
00055
00056      if (argc > 1) {
00057          infile = argv[1];
00058          if ((infilep = fopen (infile, "r")) == NULL) {
00059              fprintf (stderr, "\nERROR: Can't open file %s\n", infile);
00060              exit (EXIT_FAILURE);
00061          }
00062      }
00063      else {
00064          infilep = stdin;
00065      }
00066
00067      ix = -1;
00068
00069      while (fgets (inbuf, MAXBUF-1, infilep) != NULL) {
00070          sscanf (inbuf, "%X", &iy);
00071          if (ix == iy) fprintf (stderr, "Duplicate code point: %04X\n", ix);
00072          else ix = iy;
00073      }
00074      exit (0);
00075 }

```

5.15 src/unifont1per.c File Reference

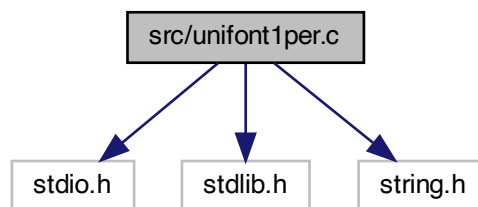
unifont1per - Read a Unifont .hex file from standard input and produce one glyph per ".bmp" bitmap file as output

```

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

```

Include dependency graph for unifont1per.c:



Macros

- #define [MAXSTRING](#) 266
- #define [MAXFILENAME](#) 20

Functions

- `int main ()`
The main function.

5.15.1 Detailed Description

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

Author

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

Copyright

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](#).

5.15.2 Macro Definition Documentation

5.15.2.1 MAXFILENAME

```
#define MAXFILENAME 20
```

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

Definition at line [60](#) of file [unifont1per.c](#).

5.15.2.2 MAXSTRING

```
#define MAXSTRING 266
```

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

Definition at line [57](#) of file [unifont1per.c](#).

5.15.3 Function Documentation

5.15.3.1 main()

int main ()

The main function.

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 69 of file [unifont1per.c](#).

```

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

```

```

00124     sscanf (instring, "%X:%s", &code_point, glyph);
00125     /* Calculate width of a glyph in pixels; 4 bits per ASCII hex digit */
00126     glyph_width = strlen (glyph) / (glyph_height / 4);
00127     snprintf (filename, MAXFILENAME, "U+%06X.bmp", code_point);
00128     header [18] = glyph_width; /* bitmap width */
00129     header [22] = -glyph_height; /* negative height --> draw top to bottom */
00130     if ((outfp = fopen (filename, "w")) != NULL) {
00131         for (i = 0; i < 62; i++) fputc (header[i], outfp);
00132         /*
00133          * Bitmap, with each row padded with zeroes if necessary
00134          * so each row is four bytes wide. (Each row must end
00135          * on a four-byte boundary, and four bytes is the maximum
00136          * possible row length for up to 32 pixels in a row.)
00137          */
00138         string_index = 0;
00139         for (i = 0; i < glyph_height; i++) {
00140             /* Read 2 ASCII hexadecimal digits (1 byte of output pixels) */
00141             sscanf (&glyph[string_index], "%2X", &nextbyte);
00142             string_index += 2;
00143             fputc (nextbyte, outfp); /* write out the 8 pixels */
00144             if (glyph_width <= 8) { /* pad row with 3 zero bytes */
00145                 fputc (0x00, outfp); fputc (0x00, outfp); fputc (0x00, outfp);
00146             }
00147             else { /* get 8 more pixels */
00148                 sscanf (&glyph[string_index], "%2X", &nextbyte);
00149                 string_index += 2;
00150                 fputc (nextbyte, outfp); /* write out the 8 pixels */
00151                 if (glyph_width <= 16) { /* pad row with 2 zero bytes */
00152                     fputc (0x00, outfp); fputc (0x00, outfp);
00153                 }
00154                 else { /* get 8 more pixels */
00155                     sscanf (&glyph[string_index], "%2X", &nextbyte);
00156                     string_index += 2;
00157                     fputc (nextbyte, outfp); /* write out the 8 pixels */
00158                     if (glyph_width <= 24) { /* pad row with 1 zero byte */
00159                         fputc (0x00, outfp);
00160                     }
00161                     else { /* get 8 more pixels */
00162                         sscanf (&glyph[string_index], "%2X", &nextbyte);
00163                         string_index += 2;
00164                         fputc (nextbyte, outfp); /* write out the 8 pixels */
00165                     } /* glyph is 32 pixels wide */
00166                 } /* glyph is 24 pixels wide */
00167             } /* glyph is 16 pixels wide */
00168         } /* glyph is 8 pixels wide */
00169         fclose (outfp);
00170     }
00171 }
00172 }
00173
00174 exit (EXIT_SUCCESS);
00175 }

```

5.16 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
00052 #include <stdio.h>
00053 #include <stdlib.h>
00054 #include <string.h>
00055
00056 /** Maximum size of an input line in a Unifont .hex file - 1. */
00057 #define MAXSTRING 266
00058
00059 /** Maximum size of a filename of the form "U+%06X.bmp". */
00060 #define MAXFILENAME 20
00061
00062 /**
00063  @brief The main function.
00064
00065  @return This program exits with status EXIT_SUCCESS.
00066 */
00067 int
00068 main () {
00069     int i; /* loop variable */
00070
00071     /*
00072      Define bitmap header bytes
00073      */
00074     unsigned char header [62] = {
00075         /*
00076          Bitmap File Header -- 14 bytes
00077          */
00078         'B', 'M', /* Signature */
00079         0x7E, 0, 0, 0, /* File Size */
00080         0, 0, 0, 0, /* Reserved */
00081         0x3E, 0, 0, 0, /* Pixel Array Offset */
00082
00083         /*
00084          Device Independent Bitmap Header -- 40 bytes
00085
00086          Image Width and Image Height are assigned final values
00087          based on the dimensions of each glyph.
00088          */
00089         0x28, 0, 0, 0, /* DIB Header Size */
00090         0x10, 0, 0, 0, /* Image Width = 16 pixels */
00091         0xF0, 0xFF, 0xFF, 0xFF, /* Image Height = -16 pixels */
00092         0x01, 0, /* Planes */
00093         0x01, 0, /* Bits Per Pixel */
00094         0, 0, 0, 0, /* Compression */
00095         0x40, 0, 0, 0, /* Image Size */
00096         0x14, 0x0B, 0, 0, /* X Pixels Per Meter = 72 dpi */
00097         0x14, 0x0B, 0, 0, /* Y Pixels Per Meter = 72 dpi */
00098         0x02, 0, 0, 0, /* Colors In Color Table */
00099     };
00100 }

```

```

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

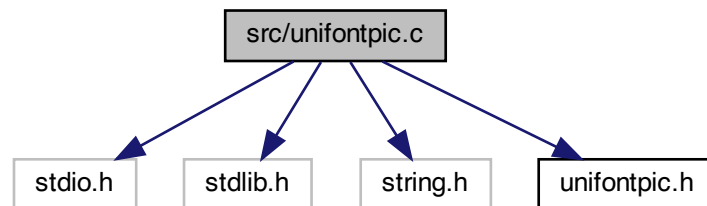
```

5.17 src/unifontpic.c File Reference

unifontpic - See the "Big Picture": the entire Unifont in one BMP bitmap

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "unifontpic.h"
```

Include dependency graph for unifontpic.c:



Macros

- `#define HDR_LEN 33`

Functions

- `int main (int argc, char **argv)`
The main function.
- `void output4 (int thisword)`
Output a 4-byte integer in little-endian order.
- `void output2 (int thisword)`
Output a 2-byte integer in little-endian order.
- `void gethex (char *instring, int plane_array[0x10000][16], int plane)`
Read a Unifont .hex-format input file from stdin.
- `void genlongbmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)`
Generate the BMP output file in long format.
- `void genwidebmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)`
Generate the BMP output file in wide format.

5.17.1 Detailed Description

unifontpic - See the "Big Picture": the entire Unifont in one BMP bitmap

Author

Paul Hardy, 2013

Copyright

Copyright (C) 2013, 2017 Paul Hardy

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

5.17.2 Macro Definition Documentation

5.17.2.1 HDR_LEN

```
#define HDR_LEN 33
```

Define length of header string for top of chart.

Definition at line [67](#) of file [unifontpic.c](#).

5.17.3 Function Documentation

5.17.3.1 genlongbmp()

```
void genlongbmp (  
    int plane_array[0x10000][16],  
    int dpi,  
    int tinynum,  
    int plane )
```

Generate the BMP output file in long format.

This function generates the BMP output file from a bitmap parameter. This is a long bitmap, 16 glyphs wide by 4,096 glyphs tall.

Parameters

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

Definition at line 294 of file [unifontpic.c](#).

```

00295 {
00296
00297     char header_string[HDR_LEN]; /* centered header */
00298     char raw_header[HDR_LEN]; /* left-aligned header */
00299     int header[16][16]; /* header row, for chart title */
00300     int hdrlen; /* length of HEADER_STRING */
00301     int startcol; /* column to start printing header, for centering */
00302
00303     unsigned leftcol[0x1000][16]; /* code point legend on left side of chart */
00304     int d1, d2, d3, d4; /* digits for filling leftcol[] legend */
00305     int codept; /* current starting code point for legend */
00306     int thisrow; /* glyph row currently being rendered */
00307     unsigned toprw[16][16]; /* code point legend on top of chart */
00308     int digitrow; /* row we're in (0..4) for the above hexdigit digits */
00309
00310     /*
00311      DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
00312      */
00313     int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */

```

```

00314 int ImageSize;
00315 int FileSize;
00316 int Width, Height; /* bitmap image width and height in pixels */
00317 int ppm; /* integer pixels per meter */
00318
00319 int i, j, k;
00320
00321 unsigned bytesout;
00322
00323 void output4(int), output2(int);
00324
00325 /*
00326  Image width and height, in pixels.
00327
00328  N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
00329 */
00330 Width = 18 * 16; /* (2 legend + 16 glyphs) * 16 pixels/glyph */
00331 Height = 4099 * 16; /* (1 header + 4096 glyphs) * 16 rows/glyph */
00332
00333 ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */
00334
00335 FileSize = DataOffset + ImageSize;
00336
00337 /* convert dots/inch to pixels/meter */
00338 if (dpi == 0) dpi = 96;
00339 ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);
00340
00341 /*
00342  Generate the BMP Header
00343 */
00344 putchar ('B');
00345 putchar ('M');
00346
00347 /*
00348  Calculate file size:
00349
00350  BMP Header + InfoHeader + Color Table + Raster Data
00351 */
00352 output4 (FileSize); /* FileSize */
00353 output4 (0x0000); /* reserved */
00354
00355 /* Calculate DataOffset */
00356 output4 (DataOffset);
00357
00358 /*
00359  InfoHeader
00360 */
00361 output4 (40); /* Size of InfoHeader */
00362 output4 (Width); /* Width of bitmap in pixels */
00363 output4 (Height); /* Height of bitmap in pixels */
00364 output2 (1); /* Planes (1 plane) */
00365 output2 (1); /* BitCount (1 = monochrome) */
00366 output4 (0); /* Compression (0 = none) */
00367 output4 (ImageSize); /* ImageSize, in bytes */
00368 output4 (ppm); /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
00369 output4 (ppm); /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
00370 output4 (2); /* ColorsUsed (= 2) */
00371 output4 (2); /* ColorsImportant (= 2) */
00372 output4 (0x00000000); /* black (reserved, B, G, R) */
00373 output4 (0x00FFFFFF); /* white (reserved, B, G, R) */
00374
00375 /*
00376  Create header row bits.
00377 */
00378 snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
00379 memset ((void *)header, 0, 16 * 16 * sizeof (int)); /* fill with white */
00380 memset ((void *)header_string, ' ', 32 * sizeof (char)); /* 32 spaces */
00381 header_string[32] = '\0'; /* null-terminated */
00382
00383 hdrlen = strlen (raw_header);
00384 if (hdrlen > 32) hdrlen = 32; /* only 32 columns to print header */
00385 startcol = 16 - ((hdrlen + 1) » 1); /* to center header */
00386 /* center up to 32 chars */
00387 memcpy (&header_string[startcol], raw_header, hdrlen);
00388
00389 /* Copy each letter's bitmap from the plane_array[] we constructed. */
00390 /* Each glyph must be single-width, to fit two glyphs in 16 pixels */
00391 for (j = 0; j < 16; j++) {
00392     for (i = 0; i < 16; i++) {
00393         header[i][j] =
00394             (ascii_bits[header_string[j+j ] & 0x7F][i] & 0xFF00) |

```

```

00395     (ascii_bits[header_string[j+j+1] & 0x7F][i] » 8);
00396     }
00397 }
00398
00399 /*
00400  Create the left column legend.
00401 */
00402 memset ((void *)leftcol, 0, 4096 * 16 * sizeof (unsigned));
00403
00404 for (codept = 0x0000; codept < 0x10000; codept += 0x10) {
00405     d1 = (codept » 12) & 0xF; /* most significant hex digit */
00406     d2 = (codept » 8) & 0xF;
00407     d3 = (codept » 4) & 0xF;
00408
00409     thisrow = codept » 4; /* rows of 16 glyphs */
00410
00411     /* fill in first and second digits */
00412     for (digitrow = 0; digitrow < 5; digitrow++) {
00413         leftcol[thisrow][2 + digitrow] =
00414             (hexdigit[d1][digitrow] « 10) |
00415             (hexdigit[d2][digitrow] « 4);
00416     }
00417
00418     /* fill in third digit */
00419     for (digitrow = 0; digitrow < 5; digitrow++) {
00420         leftcol[thisrow][9 + digitrow] = hexdigit[d3][digitrow] « 10;
00421     }
00422     leftcol[thisrow][9 + 4] |= 0xF « 4; /* underscore as 4th digit */
00423
00424     for (i = 0; i < 15; i++) {
00425         leftcol[thisrow][i] |= 0x00000002; /* right border */
00426     }
00427
00428     leftcol[thisrow][15] = 0x0000FFFE; /* bottom border */
00429
00430     if (d3 == 0xF) { /* 256-point boundary */
00431         leftcol[thisrow][15] |= 0x00FF0000; /* longer tic mark */
00432     }
00433
00434     if ((thisrow % 0x40) == 0x3F) { /* 1024-point boundary */
00435         leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
00436     }
00437 }
00438
00439 /*
00440  Create the top row legend.
00441 */
00442 memset ((void *)toprow, 0, 16 * 16 * sizeof (unsigned));
00443
00444 for (codept = 0x0; codept <= 0xF; codept++) {
00445     d1 = (codept » 12) & 0xF; /* most significant hex digit */
00446     d2 = (codept » 8) & 0xF;
00447     d3 = (codept » 4) & 0xF;
00448     d4 = codept & 0xF; /* least significant hex digit */
00449
00450     /* fill in last digit */
00451     for (digitrow = 0; digitrow < 5; digitrow++) {
00452         toprow[6 + digitrow][codept] = hexdigit[d4][digitrow] « 6;
00453     }
00454 }
00455
00456 for (j = 0; j < 16; j++) {
00457     /* force bottom pixel row to be white, for separation from glyphs */
00458     toprow[15][j] = 0x0000;
00459 }
00460
00461 /* 1 pixel row with left-hand legend line */
00462 for (j = 0; j < 16; j++) {
00463     toprow[14][j] |= 0xFFFF;
00464 }
00465
00466 /* 14 rows with line on left to fill out this character row */
00467 for (i = 13; i >= 0; i--) {
00468     for (j = 0; j < 16; j++) {
00469         toprow[i][j] |= 0x0001;
00470     }
00471 }
00472
00473 /*
00474  Now write the raster image.
00475

```

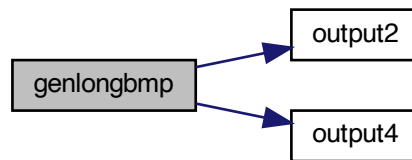
```

00476     XOR each byte with 0xFF because black = 0, white = 1 in BMP.
00477 */
00478
00479 /* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
00480 for (i = 0xFFF0; i >= 0; i -= 0x10) {
00481     thisrow = i » 4; /* 16 glyphs per row */
00482     for (j = 15; j >= 0; j--) {
00483         /* left-hand legend */
00484         putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
00485         putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
00486         putchar ((~leftcol[thisrow][j] » 8) & 0xFF);
00487         putchar (~leftcol[thisrow][j] & 0xFF);
00488         /* Unifont glyph */
00489         for (k = 0; k < 16; k++) {
00490             bytesout = ~plane_array[i+k][j] & 0xFFFF;
00491             putchar ((bytesout » 8) & 0xFF);
00492             putchar (bytesout & 0xFF);
00493         }
00494     }
00495 }
00496
00497 /*
00498  Write the top legend.
00499 */
00500 /* i == 15: bottom pixel row of header is output here */
00501 /* left-hand legend: solid black line except for right-most pixel */
00502 putchar (0x00);
00503 putchar (0x00);
00504 putchar (0x00);
00505 putchar (0x01);
00506 for (j = 0; j < 16; j++) {
00507     putchar ((~toprow[15][j] » 8) & 0xFF);
00508     putchar (~toprow[15][j] & 0xFF);
00509 }
00510
00511 putchar (0xFF);
00512 putchar (0xFF);
00513 putchar (0xFF);
00514 putchar (0xFC);
00515 for (j = 0; j < 16; j++) {
00516     putchar ((~toprow[14][j] » 8) & 0xFF);
00517     putchar (~toprow[14][j] & 0xFF);
00518 }
00519
00520 for (i = 13; i >= 0; i--) {
00521     putchar (0xFF);
00522     putchar (0xFF);
00523     putchar (0xFF);
00524     putchar (0xFD);
00525     for (j = 0; j < 16; j++) {
00526         putchar ((~toprow[i][j] » 8) & 0xFF);
00527         putchar (~toprow[i][j] & 0xFF);
00528     }
00529 }
00530
00531 /*
00532  Write the header.
00533 */
00534
00535 /* 7 completely white rows */
00536 for (i = 7; i >= 0; i--) {
00537     for (j = 0; j < 18; j++) {
00538         putchar (0xFF);
00539         putchar (0xFF);
00540     }
00541 }
00542
00543 for (i = 15; i >= 0; i--) {
00544     /* left-hand legend */
00545     putchar (0xFF);
00546     putchar (0xFF);
00547     putchar (0xFF);
00548     putchar (0xFF);
00549     /* header glyph */
00550     for (j = 0; j < 16; j++) {
00551         bytesout = ~header[i][j] & 0xFFFF;
00552         putchar ((bytesout » 8) & 0xFF);
00553         putchar (bytesout & 0xFF);
00554     }
00555 }
00556

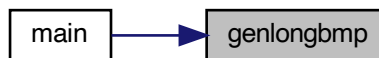
```

```
00557  /* 8 completely white rows at very top */
00558  for (i = 7; i >= 0; i--) {
00559      for (j = 0; j < 18; j++) {
00560          putchar (0xFF);
00561          putchar (0xFF);
00562      }
00563  }
00564
00565  return;
00566 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.17.3.2 genwidebmp()

```
void genwidebmp (
    int plane_array[0x10000][16],
    int dpi,
    int tinynum,
    int plane )
```

Generate the BMP output file in wide format.

This function generates the BMP output file from a bitmap parameter. This is a wide bitmap, 256 glyphs wide by 256 glyphs tall.

Parameters

in	plane_array	The array of glyph bitmaps for a plane.
in	dpi	Dots per inch, for encoding in the BMP output file header.
in	tinynum	Whether to generate tiny numbers in 256x256 grid.
in	plane	The Unicode plane, 0..17.

Definition at line 581 of file [unifontpic.c](#).

```

00582 {
00583
00584     char header_string[257];
00585     char raw_header[HDR_LEN];
00586     int header[16][256]; /* header row, for chart title */
00587     int hdrlen;          /* length of HEADER_STRING */
00588     int startcol;        /* column to start printing header, for centering */
00589
00590     unsigned leftcol[0x100][16]; /* code point legend on left side of chart */
00591     int d1, d2, d3, d4;          /* digits for filling leftcol[][] legend */
00592     int codept;                 /* current starting code point for legend */
00593     int thisrow;                /* glyph row currently being rendered */
00594     unsigned toprw[32][256];    /* code point legend on top of chart */
00595     int digitrow;               /* row we're in (0..4) for the above hexdigit digits */
00596     int hexalpha1, hexalpha2;   /* to convert hex digits to ASCII */
00597
00598     /*
00599      * DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
00600      */
00601     int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
00602     int ImageSize;
00603     int FileSize;

```

```

00604 int Width, Height; /* bitmap image width and height in pixels */
00605 int ppm; /* integer pixels per meter */
00606
00607 int i, j, k;
00608
00609 unsigned bytesout;
00610
00611 void output4(int), output2(int);
00612
00613 /*
00614  Image width and height, in pixels.
00615
00616  N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
00617 */
00618 Width = 258 * 16; /* ( 2 legend + 256 glyphs) * 16 pixels/glyph */
00619 Height = 260 * 16; /* (2 header + 2 legend + 256 glyphs) * 16 rows/glyph */
00620
00621 ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */
00622
00623 FileSize = DataOffset + ImageSize;
00624
00625 /* convert dots/inch to pixels/meter */
00626 if (dpi == 0) dpi = 96;
00627 ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);
00628
00629 /*
00630  Generate the BMP Header
00631 */
00632 putchar ('B');
00633 putchar ('M');
00634 /*
00635  Calculate file size:
00636
00637  BMP Header + InfoHeader + Color Table + Raster Data
00638 */
00639 output4 (FileSize); /* FileSize */
00640 output4 (0x0000); /* reserved */
00641 /* Calculate DataOffset */
00642 output4 (DataOffset);
00643
00644 /*
00645  InfoHeader
00646 */
00647 output4 (40); /* Size of InfoHeader */
00648 output4 (Width); /* Width of bitmap in pixels */
00649 output4 (Height); /* Height of bitmap in pixels */
00650 output2 (1); /* Planes (1 plane) */
00651 output2 (1); /* BitCount (1 = monochrome) */
00652 output4 (0); /* Compression (0 = none) */
00653 output4 (ImageSize); /* ImageSize, in bytes */
00654 output4 (ppm); /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
00655 output4 (ppm); /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
00656 output4 (2); /* ColorsUsed (= 2) */
00657 output4 (2); /* ColorsImportant (= 2) */
00658 output4 (0x00000000); /* black (reserved, B, G, R) */
00659 output4 (0x00FFFFFF); /* white (reserved, B, G, R) */
00660
00661 /*
00662  Create header row bits.
00663 */
00664 snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
00665 memset ((void *)header, 0, 256 * 16 * sizeof (int)); /* fill with white */
00666 memset ((void *)header_string, ' ', 256 * sizeof (char)); /* 256 spaces */
00667 header_string[256] = '\0'; /* null-terminated */
00668
00669 hdrlen = strlen (raw_header);
00670 /* Wide bitmap can print 256 columns, but limit to 32 columns for long bitmap. */
00671 if (hdrlen > 32) hdrlen = 32;
00672 startcol = 127 - ((hdrlen - 1) » 1); /* to center header */
00673 /* center up to 32 chars */
00674 memcpy (&header_string[startcol], raw_header, hdrlen);
00675
00676 /* Copy each letter's bitmap from the plane_array[] we constructed. */
00677 for (j = 0; j < 256; j++) {
00678     for (i = 0; i < 16; i++) {
00679         header[i][j] = ascii_bits[header_string[j] & 0x7F][i];
00680     }
00681 }
00682
00683 /*
00684  Create the left column legend.

```

```

00685 */
00686 memset ((void *)leftcol, 0, 256 * 16 * sizeof (unsigned));
00687
00688 for (codept = 0x0000; codept < 0x10000; codept += 0x100) {
00689     d1 = (codept » 12) & 0xF; /* most significant hex digit */
00690     d2 = (codept » 8) & 0xF;
00691
00692     thisrow = codept » 8; /* rows of 256 glyphs */
00693
00694     /* fill in first and second digits */
00695
00696     if (tinynum) { /* use 4x5 pixel glyphs */
00697         for (digitrow = 0; digitrow < 5; digitrow++) {
00698             leftcol[thisrow][6 + digitrow] =
00699                 (hexdigit[d1][digitrow] « 10) |
00700                 (hexdigit[d2][digitrow] « 4);
00701         }
00702     }
00703     else { /* bigger numbers -- use glyphs from Unifont itself */
00704         /* convert hexadecimal digits to ASCII equivalent */
00705         hexalpha1 = d1 < 0xA ? '0' + d1 : 'A' + d1 - 0xA;
00706         hexalpha2 = d2 < 0xA ? '0' + d2 : 'A' + d2 - 0xA;
00707
00708         for (i = 0 ; i < 16; i++) {
00709             leftcol[thisrow][i] =
00710                 (ascii_bits[hexalpha1][i] « 2) |
00711                 (ascii_bits[hexalpha2][i] » 6);
00712         }
00713     }
00714
00715     for (i = 0; i < 15; i++) {
00716         leftcol[thisrow][i] |= 0x00000002; /* right border */
00717     }
00718
00719     leftcol[thisrow][15] = 0x0000FFFE; /* bottom border */
00720
00721     if (d2 == 0xF) { /* 4096-point boundary */
00722         leftcol[thisrow][15] |= 0x00FF0000; /* longer tic mark */
00723     }
00724
00725     if ((thisrow % 0x40) == 0x3F) { /* 16,384-point boundary */
00726         leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
00727     }
00728 }
00729
00730 /*
00731  Create the top row legend.
00732 */
00733 memset ((void *)toprow, 0, 32 * 256 * sizeof (unsigned));
00734
00735 for (codept = 0x00; codept <= 0xFF; codept++) {
00736     d3 = (codept » 4) & 0xF;
00737     d4 = codept & 0xF; /* least significant hex digit */
00738
00739     if (tinynum) {
00740         for (digitrow = 0; digitrow < 5; digitrow++) {
00741             toprow[16 + 6 + digitrow][codept] =
00742                 (hexdigit[d3][digitrow] « 10) |
00743                 (hexdigit[d4][digitrow] « 4);
00744         }
00745     }
00746     else {
00747         /* convert hexadecimal digits to ASCII equivalent */
00748         hexalpha1 = d3 < 0xA ? '0' + d3 : 'A' + d3 - 0xA;
00749         hexalpha2 = d4 < 0xA ? '0' + d4 : 'A' + d4 - 0xA;
00750         for (i = 0 ; i < 16; i++) {
00751             toprow[14 + i][codept] =
00752                 (ascii_bits[hexalpha1][i] ) |
00753                 (ascii_bits[hexalpha2][i] » 7);
00754         }
00755     }
00756 }
00757
00758 for (j = 0; j < 256; j++) {
00759     /* force bottom pixel row to be white, for separation from glyphs */
00760     toprow[16 + 15][j] = 0x0000;
00761 }
00762
00763 /* 1 pixel row with left-hand legend line */
00764 for (j = 0; j < 256; j++) {
00765     toprow[16 + 14][j] |= 0xFFFF;

```



```

00766 }
00767
00768 /* 14 rows with line on left to fill out this character row */
00769 for (i = 13; i >= 0; i--) {
00770     for (j = 0; j < 256; j++) {
00771         toprow[16 + i][j] |= 0x0001;
00772     }
00773 }
00774
00775 /* Form the longer tic marks in top legend */
00776 for (i = 8; i < 16; i++) {
00777     for (j = 0x0F; j < 0x100; j += 0x10) {
00778         toprow[i][j] |= 0x0001;
00779     }
00780 }
00781
00782 /*
00783     Now write the raster image.
00784
00785     XOR each byte with 0xFF because black = 0, white = 1 in BMP.
00786 */
00787
00788 /* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
00789 for (i = 0xFF00; i >= 0; i -= 0x100) {
00790     thisrow = i » 8; /* 256 glyphs per row */
00791     for (j = 15; j >= 0; j--) {
00792         /* left-hand legend */
00793         putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
00794         putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
00795         putchar ((~leftcol[thisrow][j] » 8) & 0xFF);
00796         putchar (~leftcol[thisrow][j] & 0xFF);
00797         /* Unifont glyph */
00798         for (k = 0x00; k < 0x100; k++) {
00799             bytesout = ~plane_array[i+k][j] & 0xFFFF;
00800             putchar ((bytesout » 8) & 0xFF);
00801             putchar (bytesout & 0xFF);
00802         }
00803     }
00804 }
00805
00806 /*
00807     Write the top legend.
00808 */
00809 /* i == 15: bottom pixel row of header is output here */
00810 /* left-hand legend: solid black line except for right-most pixel */
00811 putchar (0x00);
00812 putchar (0x00);
00813 putchar (0x00);
00814 putchar (0x01);
00815 for (j = 0; j < 256; j++) {
00816     putchar ((~toprow[16 + 15][j] » 8) & 0xFF);
00817     putchar (~toprow[16 + 15][j] & 0xFF);
00818 }
00819
00820 putchar (0xFF);
00821 putchar (0xFF);
00822 putchar (0xFF);
00823 putchar (0xFC);
00824 for (j = 0; j < 256; j++) {
00825     putchar ((~toprow[16 + 14][j] » 8) & 0xFF);
00826     putchar (~toprow[16 + 14][j] & 0xFF);
00827 }
00828
00829 for (i = 16 + 13; i >= 0; i--) {
00830     if (i >= 8) { /* make vertical stroke on right */
00831         putchar (0xFF);
00832         putchar (0xFF);
00833         putchar (0xFF);
00834         putchar (0xFD);
00835     }
00836     else { /* all white */
00837         putchar (0xFF);
00838         putchar (0xFF);
00839         putchar (0xFF);
00840         putchar (0xFF);
00841     }
00842     for (j = 0; j < 256; j++) {
00843         putchar ((~toprow[i][j] » 8) & 0xFF);
00844         putchar (~toprow[i][j] & 0xFF);
00845     }
00846 }

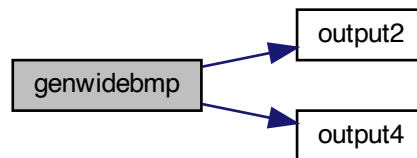
```

```

00847
00848  /*
00849  Write the header.
00850  */
00851
00852  /* 8 completely white rows */
00853  for (i = 7; i >= 0; i--) {
00854      for (j = 0; j < 258; j++) {
00855          putchar (0xFF);
00856          putchar (0xFF);
00857      }
00858  }
00859
00860  for (i = 15; i >= 0; i--) {
00861      /* left-hand legend */
00862      putchar (0xFF);
00863      putchar (0xFF);
00864      putchar (0xFF);
00865      putchar (0xFF);
00866      /* header glyph */
00867      for (j = 0; j < 256; j++) {
00868          bytesout = ~header[i][j] & 0xFFFF;
00869          putchar ((bytesout » 8) & 0xFF);
00870          putchar ( bytesout      & 0xFF);
00871      }
00872  }
00873
00874  /* 8 completely white rows at very top */
00875  for (i = 7; i >= 0; i--) {
00876      for (j = 0; j < 258; j++) {
00877          putchar (0xFF);
00878          putchar (0xFF);
00879      }
00880  }
00881
00882  return;
00883 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



5.17.3.3 gethex()

```
void gethex (
    char * instring,
    int plane_array[0x10000][16],
    int plane )
```

Read a Unifont .hex-format input file from stdin.

Each glyph can be 2, 4, 6, or 8 ASCII hexadecimal digits wide. [Glyph](#) height is fixed at 16 pixels.

Parameters

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

Definition at line 215 of file unifontpic.c.

```
00216 {
00217     char *bitstring; /* pointer into instring for glyph bitmap */
00218     int i;           /* loop variable */
00219     int codept; /* the Unicode code point of the current glyph */
00220     int glyph_plane; /* Unicode plane of current glyph */
00221     int ndigits; /* number of ASCII hexadecimal digits in glyph */
00222     int bytespl; /* bytes per line of pixels in a glyph */
00223     int temprow; /* 1 row of a quadruple-width glyph */
00224     int newrow; /* 1 row of double-width output pixels */
00225     unsigned bitmask; /* to mask off 2 bits of long width glyph */
00226
00227     /*
00228     Read each input line and place its glyph into the bit array.
00229     */
00230     sscanf (instring, "%X", &codept);
00231     glyph_plane = codept » 16;
00232     if (glyph_plane == plane) {
00233         codept &= 0xFFFF; /* array index will only have 16 bit address */
00234         /* find the colon separator */
00235         for (i = 0; (i < 9) && (instring[i] != ':'); i++);
00236         i++; /* position past it */
00237         bitstring = &instring[i];
```

```

00238     ndigits = strlen (bitstring);
00239     /* don't count '\n' at end of line if present */
00240     if (bitstring[ndigits - 1] == '\n') ndigits--;
00241     bytespl = ndigits » 5; /* 16 rows per line, 2 digits per byte */
00242
00243     if (bytespl >= 1 && bytespl <= 4) {
00244         for (i = 0; i < 16; i++) { /* 16 rows per glyph */
00245             /* Read correct number of hexadecimal digits given glyph width */
00246             switch (bytespl) {
00247                 case 1: sscanf (bitstring, "%2X", &temprow);
00248                     bitstring += 2;
00249                     temprow «= 8; /* left-justify single-width glyph */
00250                     break;
00251                 case 2: sscanf (bitstring, "%4X", &temprow);
00252                     bitstring += 4;
00253                     break;
00254                 /* cases 3 and 4 widths will be compressed by 50% (see below) */
00255                 case 3: sscanf (bitstring, "%6X", &temprow);
00256                     bitstring += 6;
00257                     temprow «= 8; /* left-justify */
00258                     break;
00259                 case 4: sscanf (bitstring, "%8X", &temprow);
00260                     bitstring += 8;
00261                     break;
00262             } /* switch on number of bytes per row */
00263             /* compress glyph width by 50% if greater than double-width */
00264             if (bytespl > 2) {
00265                 newrow = 0x0000;
00266                 /* mask off 2 bits at a time to convert each pair to 1 bit out */
00267                 for (bitmask = 0xC0000000; bitmask != 0; bitmask »= 2) {
00268                     newrow «= 1;
00269                     if ((temprow & bitmask) != 0) newrow |= 1;
00270                 }
00271                 temprow = newrow;
00272             } /* done conditioning glyphs beyond double-width */
00273             plane_array[codept][i] = temprow; /* store glyph bitmap for output */
00274         } /* for each row */
00275     } /* if 1 to 4 bytes per row/line */
00276 } /* if this is the plane we are seeking */
00277
00278     return;
00279 }

```

Here is the caller graph for this function:



5.17.3.4 main()

```

int main (
    int argc,
    char ** argv )

```

The main function.

Parameters

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

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 87 of file [unifontpic.c](#).

```

00088 {
00089     /* Input line buffer */
00090     char instring[MAXSTRING];
00091
00092     /* long and dpi are set from command-line options */
00093     int wide=1; /* =1 for a 256x256 grid, =0 for a 16x4096 grid */
00094     int dpi=96; /* change for 256x256 grid to fit paper if desired */
00095     int tinynum=0; /* whether to use tiny labels for 256x256 grid */
00096
00097     int i, j; /* loop variables */
00098
00099     int plane=0; /* Unicode plane, 0..17; Plane 0 is default */
00100     /* 16 pixel rows for each of 65,536 glyphs in a Unicode plane */
00101     int plane_array[0x10000][16];
00102
00103     void gethex();
00104     void genlongbmp();
00105     void genwidebmp();
00106
00107     if (argc > 1) {
00108         for (i = 1; i < argc; i++) {
00109             if (strcmp (argv[i], "-l", 2) == 0) { /* long display */
00110                 wide = 0;
00111             }
00112             else if (strcmp (argv[i], "-d", 2) == 0) {
00113                 dpi = atoi (&argv[i][2]); /* dots/inch specified on command line */
00114             }
00115             else if (strcmp (argv[i], "-t", 2) == 0) {
00116                 tinynum = 1;
00117             }
00118             else if (strcmp (argv[i], "-P", 2) == 0) {
00119                 /* Get Unicode plane */
00120                 for (j = 2; argv[i][j] != '\0'; j++) {
00121                     if (argv[i][j] < '0' || argv[i][j] > '9') {
00122                         fprintf (stderr,
00123                             "ERROR: Specify Unicode plane as decimal number.\n\n");
00124                         exit (EXIT_FAILURE);
00125                     }
00126                 }
00127                 plane = atoi (&argv[i][2]); /* Unicode plane, 0..17 */
00128                 if (plane < 0 || plane > 17) {
00129                     fprintf (stderr,
00130                         "ERROR: Plane out of Unicode range [0,17].\n\n");

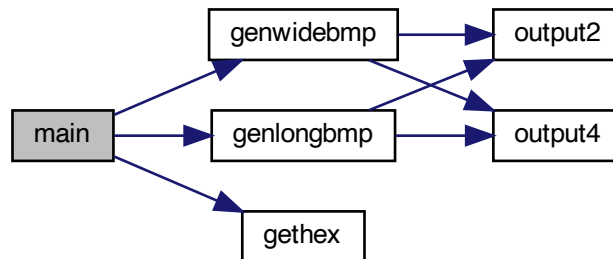
```

```

00131         exit (EXIT_FAILURE);
00132     }
00133 }
00134 }
00135 }
00136
00137
00138 /*
00139  Initialize the ASCII bitmap array for chart titles
00140 */
00141 for (i = 0; i < 128; i++) {
00142     gethex (ascii_hex[i], plane_array, 0); /* convert Unifont hexadecimal string to bitmap */
00143     for (j = 0; j < 16; j++) ascii_bits[i][j] = plane_array[i][j];
00144 }
00145
00146
00147 /*
00148  Read in the Unifont hex file to render from standard input
00149 */
00150 memset ((void *)plane_array, 0, 0x10000 * 16 * sizeof (int));
00151 while (fgets (instring, MAXSTRING, stdin) != NULL) {
00152     gethex (instring, plane_array, plane); /* read .hex input file and fill plane_array with glyph data */
00153 } /* while not EOF */
00154
00155
00156 /*
00157  Write plane_array glyph data to BMP file as wide or long bitmap.
00158 */
00159 if (wide) {
00160     genwidebmp (plane_array, dpi, tinyum, plane);
00161 }
00162 else {
00163     genlongbmp (plane_array, dpi, tinyum, plane);
00164 }
00165
00166 exit (EXIT_SUCCESS);
00167 }

```

Here is the call graph for this function:



5.17.3.5 output2()

```

void output2 (
    int thisword )

```

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

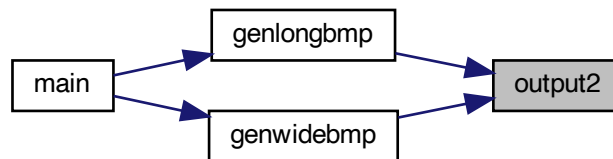
Parameters

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

Definition at line 194 of file [unifontpic.c](#).

```
00195 {  
00196  
00197     putchar ( thisword      & 0xFF);  
00198     putchar ((thisword » 8) & 0xFF);  
00199  
00200     return;  
00201 }
```

Here is the caller graph for this function:



5.17.3.6 output4()

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

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

Parameters

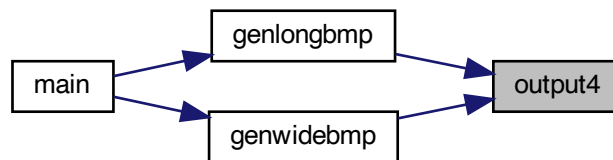
in	thisword	The 4-byte integer to output as binary data.
----	----------	--

Definition at line 176 of file [unifontpic.c](#).

```

00177 {
00178
00179     putchar ( thisword      & 0xFF);
00180     putchar ((thisword » 8) & 0xFF);
00181     putchar ((thisword » 16) & 0xFF);
00182     putchar ((thisword » 24) & 0xFF);
00183
00184     return;
00185 }
```

Here is the caller graph for this function:



5.18 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
00060
00061 #include <stdio.h>
00062 #include <stdlib.h>
00063 #include <string.h>
00064 #include "unifontpic.h"
00065
00066 /** Define length of header string for top of chart. */
00067 #define HDR_LEN 33
00068
00069
00070 /*
00071     Stylistic Note:
00072
00073     Many variables in this program use multiple words scrunched
00074     together, with each word starting with an upper-case letter.
00075     This is only done to match the canonical field names in the
00076     Windows Bitmap Graphics spec.
00077 */
00078
00079 /**
00080     @brief The main function.
00081
00082     @param[in] argc The count of command line arguments.
00083     @param[in] argv Pointer to array of command line arguments.
00084     @return This program exits with status EXIT_SUCCESS.
00085 */
00086 int
00087 main (int argc, char **argv)
00088 {
00089     /* Input line buffer */
00090     char instring[MAXSTRING];
00091
00092     /* long and dpi are set from command-line options */
00093     int wide=1; /* =1 for a 256x256 grid, =0 for a 16x4096 grid */
00094     int dpi=96; /* change for 256x256 grid to fit paper if desired */
00095     int tinynum=0; /* whether to use tiny labels for 256x256 grid */
00096
00097     int i, j; /* loop variables */
00098
00099     int plane=0; /* Unicode plane, 0..17; Plane 0 is default */
00100     /* 16 pixel rows for each of 65,536 glyphs in a Unicode plane */
00101     int plane_array[0x10000][16];

```

```

00102
00103 void gethex();
00104 void genlongbmp();
00105 void genwidebmp();
00106
00107 if (argc > 1) {
00108     for (i = 1; i < argc; i++) {
00109         if (strcmp (argv[i], "-l", 2) == 0) { /* long display */
00110             wide = 0;
00111         }
00112         else if (strcmp (argv[i], "-d", 2) == 0) {
00113             dpi = atoi (&argv[i][2]); /* dots/inch specified on command line */
00114         }
00115         else if (strcmp (argv[i], "-t", 2) == 0) {
00116             tinynum = 1;
00117         }
00118         else if (strcmp (argv[i], "-P", 2) == 0) {
00119             /* Get Unicode plane */
00120             for (j = 2; argv[i][j] != '\0'; j++) {
00121                 if (argv[i][j] < '0' || argv[i][j] > '9') {
00122                     fprintf (stderr,
00123                             "ERROR: Specify Unicode plane as decimal number.\n\n");
00124                     exit (EXIT_FAILURE);
00125                 }
00126             }
00127             plane = atoi (&argv[i][2]); /* Unicode plane, 0..17 */
00128             if (plane < 0 || plane > 17) {
00129                 fprintf (stderr,
00130                         "ERROR: Plane out of Unicode range [0,17].\n\n");
00131                 exit (EXIT_FAILURE);
00132             }
00133         }
00134     }
00135 }
00136
00137
00138 /*
00139  * Initialize the ASCII bitmap array for chart titles
00140  */
00141 for (i = 0; i < 128; i++) {
00142     gethex (ascii_hex[i], plane_array, 0); /* convert Unifont hexadecimal string to bitmap */
00143     for (j = 0; j < 16; j++) ascii_bits[i][j] = plane_array[i][j];
00144 }
00145
00146
00147 /*
00148  * Read in the Unifont hex file to render from standard input
00149  */
00150 memset ((void *)plane_array, 0, 0x10000 * 16 * sizeof (int));
00151 while (fgets (instr, MAXSTRING, stdin) != NULL) {
00152     gethex (instr, plane_array, plane); /* read .hex input file and fill plane_array with glyph data */
00153 } /* while not EOF */
00154
00155
00156 /*
00157  * Write plane_array glyph data to BMP file as wide or long bitmap.
00158  */
00159 if (wide) {
00160     genwidebmp (plane_array, dpi, tinynum, plane);
00161 }
00162 else {
00163     genlongbmp (plane_array, dpi, tinynum, plane);
00164 }
00165
00166 exit (EXIT_SUCCESS);
00167 }
00168
00169
00170 /**
00171  * @brief Output a 4-byte integer in little-endian order.
00172  *
00173  * @param[in] thisword The 4-byte integer to output as binary data.
00174  */
00175 void
00176 output4 (int thisword)
00177 {
00178
00179     putchar (thisword & 0xFF);
00180     putchar ((thisword >> 8) & 0xFF);
00181     putchar ((thisword >> 16) & 0xFF);
00182     putchar ((thisword >> 24) & 0xFF);

```

```

00183
00184     return;
00185 }
00186
00187
00188 /**
00189  @brief Output a 2-byte integer in little-endian order.
00190
00191  @param[in] thisword The 2-byte integer to output as binary data.
00192 */
00193 void
00194 output2 (int thisword)
00195 {
00196     putchar ( thisword      & 0xFF);
00197     putchar ((thisword » 8) & 0xFF);
00198
00199     return;
00200 }
00201
00202
00203
00204 /**
00205  @brief Read a Unifont .hex-format input file from stdin.
00206
00207  Each glyph can be 2, 4, 6, or 8 ASCII hexadecimal digits wide.
00208  Glyph height is fixed at 16 pixels.
00209
00210  @param[in] instring One line from a Unifont .hex-format file.
00211  @param[in,out] plane_array Bitmap for this plane, one bitmap row per element.
00212  @param[in] plane The Unicode plane, 0..17.
00213 */
00214 void
00215 gethex (char *instring, int plane_array[0x10000][16], int plane)
00216 {
00217     char *bitstring; /* pointer into instring for glyph bitmap */
00218     int i; /* loop variable */
00219     int codept; /* the Unicode code point of the current glyph */
00220     int glyph_plane; /* Unicode plane of current glyph */
00221     int ndigits; /* number of ASCII hexadecimal digits in glyph */
00222     int bytespl; /* bytes per line of pixels in a glyph */
00223     int temprow; /* 1 row of a quadruple-width glyph */
00224     int newrow; /* 1 row of double-width output pixels */
00225     unsigned bitmask; /* to mask off 2 bits of long width glyph */
00226
00227     /*
00228      Read each input line and place its glyph into the bit array.
00229     */
00230     sscanf (instring, "%X", &codept);
00231     glyph_plane = codept » 16;
00232     if (glyph_plane == plane) {
00233         codept &= 0xFFFF; /* array index will only have 16 bit address */
00234         /* find the colon separator */
00235         for (i = 0; (i < 9) && (instring[i] != ':'); i++);
00236         i++; /* position past it */
00237         bitstring = &instring[i];
00238         ndigits = strlen (bitstring);
00239         /* don't count '\n' at end of line if present */
00240         if (bitstring[ndigits - 1] == '\n') ndigits--;
00241         bytespl = ndigits » 5; /* 16 rows per line, 2 digits per byte */
00242
00243         if (bytespl >= 1 && bytespl <= 4) {
00244             for (i = 0; i < 16; i++) { /* 16 rows per glyph */
00245                 /* Read correct number of hexadecimal digits given glyph width */
00246                 switch (bytespl) {
00247                     case 1: sscanf (bitstring, "%2X", &temprow);
00248                             bitstring += 2;
00249                             temprow «= 8; /* left-justify single-width glyph */
00250                             break;
00251                     case 2: sscanf (bitstring, "%4X", &temprow);
00252                             bitstring += 4;
00253                             break;
00254                     /* cases 3 and 4 widths will be compressed by 50% (see below) */
00255                     case 3: sscanf (bitstring, "%6X", &temprow);
00256                             bitstring += 6;
00257                             temprow «= 8; /* left-justify */
00258                             break;
00259                     case 4: sscanf (bitstring, "%8X", &temprow);
00260                             bitstring += 8;
00261                             break;
00262                 } /* switch on number of bytes per row */
00263                 /* compress glyph width by 50% if greater than double-width */

```

```

00264         if (bytespl > 2) {
00265             newrow = 0x0000;
00266             /* mask off 2 bits at a time to convert each pair to 1 bit out */
00267             for (bitmask = 0xC0000000; bitmask != 0; bitmask »= 2) {
00268                 newrow «= 1;
00269                 if ((temprow & bitmask) != 0) newrow |= 1;
00270             }
00271             temprow = newrow;
00272         } /* done conditioning glyphs beyond double-width */
00273         plane_array[codept][i] = temprow; /* store glyph bitmap for output */
00274     } /* for each row */
00275 } /* if 1 to 4 bytes per row/line */
00276 } /* if this is the plane we are seeking */
00277
00278 return;
00279 }
00280
00281
00282 /**
00283  * @brief Generate the BMP output file in long format.
00284  *
00285  * This function generates the BMP output file from a bitmap parameter.
00286  * This is a long bitmap, 16 glyphs wide by 4,096 glyphs tall.
00287  *
00288  * @param[in] plane_array The array of glyph bitmaps for a plane.
00289  * @param[in] dpi Dots per inch, for encoding in the BMP output file header.
00290  * @param[in] tinynum Whether to generate tiny numbers in wide grid (unused).
00291  * @param[in] plane The Unicode plane, 0..17.
00292  */
00293 void
00294 genlongbmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)
00295 {
00296     char header_string[HDR_LEN]; /* centered header */
00297     char raw_header[HDR_LEN]; /* left-aligned header */
00298     int header[16][16]; /* header row, for chart title */
00299     int hdrlen; /* length of HEADER_STRING */
00300     int startcol; /* column to start printing header, for centering */
00301
00302     unsigned leftcol[0x1000][16]; /* code point legend on left side of chart */
00303     int d1, d2, d3, d4; /* digits for filling leftcol[][] legend */
00304     int codept; /* current starting code point for legend */
00305     int thisrow; /* glyph row currently being rendered */
00306     unsigned toprow[16][16]; /* code point legend on top of chart */
00307     int digitrow; /* row we're in (0..4) for the above hexdigit digits */
00308
00309     /*
00310      * DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
00311      */
00312     int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
00313     int ImageSize;
00314     int FileSize;
00315     int Width, Height; /* bitmap image width and height in pixels */
00316     int ppm; /* integer pixels per meter */
00317
00318     int i, j, k;
00319
00320     unsigned bytesout;
00321
00322     void output4(int), output2(int);
00323
00324     /*
00325      * Image width and height, in pixels.
00326      *
00327      * N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
00328      */
00329     Width = 18 * 16; /* (2 legend + 16 glyphs) * 16 pixels/glyph */
00330     Height = 4099 * 16; /* (1 header + 4096 glyphs) * 16 rows/glyph */
00331
00332     ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */
00333
00334     FileSize = DataOffset + ImageSize;
00335
00336     /* convert dots/inch to pixels/meter */
00337     if (dpi == 0) dpi = 96;
00338     ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);
00339
00340     /*
00341      * Generate the BMP Header
00342      */
00343     putchar ('B');
00344

```

```

00345 putchar ('M');
00346
00347 /*
00348 Calculate file size:
00349
00350 BMP Header + InfoHeader + Color Table + Raster Data
00351 */
00352 output4 (FileSize); /* FileSize */
00353 output4 (0x0000); /* reserved */
00354
00355 /* Calculate DataOffset */
00356 output4 (DataOffset);
00357
00358 /*
00359 InfoHeader
00360 */
00361 output4 (40); /* Size of InfoHeader */
00362 output4 (Width); /* Width of bitmap in pixels */
00363 output4 (Height); /* Height of bitmap in pixels */
00364 output2 (1); /* Planes (1 plane) */
00365 output2 (1); /* BitCount (1 = monochrome) */
00366 output4 (0); /* Compression (0 = none) */
00367 output4 (ImageSize); /* ImageSize, in bytes */
00368 output4 (ppm); /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
00369 output4 (ppm); /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
00370 output4 (2); /* ColorsUsed (= 2) */
00371 output4 (2); /* ColorsImportant (= 2) */
00372 output4 (0x00000000); /* black (reserved, B, G, R) */
00373 output4 (0x00FFFFFF); /* white (reserved, B, G, R) */
00374
00375 /*
00376 Create header row bits.
00377 */
00378 snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
00379 memset ((void *)header, 0, 16 * 16 * sizeof (int)); /* fill with white */
00380 memset ((void *)header_string, ' ', 32 * sizeof (char)); /* 32 spaces */
00381 header_string[32] = '\0'; /* null-terminated */
00382
00383 hdrlen = strlen (raw_header);
00384 if (hdrlen > 32) hdrlen = 32; /* only 32 columns to print header */
00385 startcol = 16 - ((hdrlen + 1) » 1); /* to center header */
00386 /* center up to 32 chars */
00387 memcpy (&header_string[startcol], raw_header, hdrlen);
00388
00389 /* Copy each letter's bitmap from the plane_array[] we constructed. */
00390 /* Each glyph must be single-width, to fit two glyphs in 16 pixels */
00391 for (j = 0; j < 16; j++) {
00392     for (i = 0; i < 16; i++) {
00393         header[i][j] =
00394             (ascii_bits[header_string[j+i] ] & 0x7F)[i] & 0xFF00 |
00395             (ascii_bits[header_string[j+i+1] & 0x7F)[i] » 8);
00396     }
00397 }
00398
00399 /*
00400 Create the left column legend.
00401 */
00402 memset ((void *)leftcol, 0, 4096 * 16 * sizeof (unsigned));
00403
00404 for (codept = 0x0000; codept < 0x10000; codept += 0x10) {
00405     d1 = (codept » 12) & 0xF; /* most significant hex digit */
00406     d2 = (codept » 8) & 0xF;
00407     d3 = (codept » 4) & 0xF;
00408
00409     thisrow = codept » 4; /* rows of 16 glyphs */
00410
00411     /* fill in first and second digits */
00412     for (digitrow = 0; digitrow < 5; digitrow++) {
00413         leftcol[thisrow][2 + digitrow] =
00414             (hexdigit[d1][digitrow] « 10) |
00415             (hexdigit[d2][digitrow] « 4);
00416     }
00417
00418     /* fill in third digit */
00419     for (digitrow = 0; digitrow < 5; digitrow++) {
00420         leftcol[thisrow][9 + digitrow] = hexdigit[d3][digitrow] « 10;
00421     }
00422     leftcol[thisrow][9 + 4] |= 0xF « 4; /* underscore as 4th digit */
00423
00424     for (i = 0; i < 15; i++) {
00425         leftcol[thisrow][i] |= 0x00000002; /* right border */

```

```

00426     }
00427
00428     leftcol[thisrow][15] = 0x000FFFE;      /* bottom border */
00429
00430     if (d3 == 0xF) {                      /* 256-point boundary */
00431         leftcol[thisrow][15] |= 0x0FF0000; /* longer tic mark */
00432     }
00433
00434     if ((thisrow % 0x40) == 0x3F) {       /* 1024-point boundary */
00435         leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
00436     }
00437 }
00438
00439 /*
00440  Create the top row legend.
00441 */
00442 memset ((void *)toprow, 0, 16 * 16 * sizeof (unsigned));
00443
00444 for (codept = 0x0; codept <= 0xF; codept++) {
00445     d1 = (codept » 12) & 0xF; /* most significant hex digit */
00446     d2 = (codept » 8) & 0xF;
00447     d3 = (codept » 4) & 0xF;
00448     d4 = codept & 0xF; /* least significant hex digit */
00449
00450     /* fill in last digit */
00451     for (digitrow = 0; digitrow < 5; digitrow++) {
00452         toprow[6 + digitrow][codept] = hexdigit[d4][digitrow] « 6;
00453     }
00454 }
00455
00456 for (j = 0; j < 16; j++) {
00457     /* force bottom pixel row to be white, for separation from glyphs */
00458     toprow[15][j] = 0x0000;
00459 }
00460
00461 /* 1 pixel row with left-hand legend line */
00462 for (j = 0; j < 16; j++) {
00463     toprow[14][j] |= 0xFFFF;
00464 }
00465
00466 /* 14 rows with line on left to fill out this character row */
00467 for (i = 13; i >= 0; i--) {
00468     for (j = 0; j < 16; j++) {
00469         toprow[i][j] |= 0x0001;
00470     }
00471 }
00472
00473 /*
00474  Now write the raster image.
00475
00476  XOR each byte with 0xFF because black = 0, white = 1 in BMP.
00477 */
00478
00479 /* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
00480 for (i = 0xFFF0; i >= 0; i -= 0x10) {
00481     thisrow = i » 4; /* 16 glyphs per row */
00482     for (j = 15; j >= 0; j--) {
00483         /* left-hand legend */
00484         putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
00485         putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
00486         putchar ((~leftcol[thisrow][j] » 8) & 0xFF);
00487         putchar (~leftcol[thisrow][j] & 0xFF);
00488         /* Unifont glyph */
00489         for (k = 0; k < 16; k++) {
00490             bytesout = ~plane_array[i+k][j] & 0xFFFF;
00491             putchar ((bytesout » 8) & 0xFF);
00492             putchar ( bytesout & 0xFF);
00493         }
00494     }
00495 }
00496
00497 /*
00498  Write the top legend.
00499 */
00500 /* i == 15: bottom pixel row of header is output here */
00501 /* left-hand legend: solid black line except for right-most pixel */
00502 putchar (0x00);
00503 putchar (0x00);
00504 putchar (0x00);
00505 putchar (0x01);
00506 for (j = 0; j < 16; j++) {

```

```

00507     putchar ((~toprow[15][j] » 8) & 0xFF);
00508     putchar ( ~toprow[15][j]      & 0xFF);
00509 }
00510
00511 putchar (0xFF);
00512 putchar (0xFF);
00513 putchar (0xFF);
00514 putchar (0xFC);
00515 for (j = 0; j < 16; j++) {
00516     putchar ((~toprow[14][j] » 8) & 0xFF);
00517     putchar ( ~toprow[14][j]      & 0xFF);
00518 }
00519
00520 for (i = 13; i >= 0; i--) {
00521     putchar (0xFF);
00522     putchar (0xFF);
00523     putchar (0xFF);
00524     putchar (0xFD);
00525     for (j = 0; j < 16; j++) {
00526         putchar ((~toprow[i][j] » 8) & 0xFF);
00527         putchar ( ~toprow[i][j]      & 0xFF);
00528     }
00529 }
00530
00531 /*
00532  Write the header.
00533 */
00534
00535 /* 7 completely white rows */
00536 for (i = 7; i >= 0; i--) {
00537     for (j = 0; j < 18; j++) {
00538         putchar (0xFF);
00539         putchar (0xFF);
00540     }
00541 }
00542
00543 for (i = 15; i >= 0; i--) {
00544     /* left-hand legend */
00545     putchar (0xFF);
00546     putchar (0xFF);
00547     putchar (0xFF);
00548     putchar (0xFF);
00549     /* header glyph */
00550     for (j = 0; j < 16; j++) {
00551         bytesout = ~header[i][j] & 0xFFFF;
00552         putchar ((bytesout » 8) & 0xFF);
00553         putchar ( bytesout      & 0xFF);
00554     }
00555 }
00556
00557 /* 8 completely white rows at very top */
00558 for (i = 7; i >= 0; i--) {
00559     for (j = 0; j < 18; j++) {
00560         putchar (0xFF);
00561         putchar (0xFF);
00562     }
00563 }
00564
00565 return;
00566 }
00567
00568 /**
00569  @brief Generate the BMP output file in wide format.
00570
00571  This function generates the BMP output file from a bitmap parameter.
00572  This is a wide bitmap, 256 glyphs wide by 256 glyphs tall.
00573
00574  @param[in] plane_array The array of glyph bitmaps for a plane.
00575  @param[in] dpi Dots per inch, for encoding in the BMP output file header.
00576  @param[in] tinynum Whether to generate tiny numbers in 256x256 grid.
00577  @param[in] plane The Unicode plane, 0..17.
00578 */
00579 void
00580 genwidebmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)
00581 {
00582     char header_string[257];
00583     char raw_header[HDR_LEN];
00584     int header[16][256]; /* header row, for chart title */
00585     int hdrlen;          /* length of HEADER_STRING */

```

```

00588 int startcol;      /* column to start printing header, for centering */
00589
00590 unsigned leftcol[0x100][16]; /* code point legend on left side of chart */
00591 int d1, d2, d3, d4;      /* digits for filling leftcol[] legend */
00592 int codept;             /* current starting code point for legend */
00593 int thisrow;            /* glyph row currently being rendered */
00594 unsigned toprow[32][256]; /* code point legend on top of chart */
00595 int digitrow;           /* row we're in (0..4) for the above hexdigit digits */
00596 int hexalpha1, hexalpha2; /* to convert hex digits to ASCII */
00597
00598 /*
00599  DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
00600 */
00601 int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
00602 int ImageSize;
00603 int FileSize;
00604 int Width, Height; /* bitmap image width and height in pixels */
00605 int ppm;           /* integer pixels per meter */
00606
00607 int i, j, k;
00608
00609 unsigned bytesout;
00610
00611 void output4(int), output2(int);
00612
00613 /*
00614  Image width and height, in pixels.
00615
00616  N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
00617 */
00618 Width = 258 * 16; /* ( 2 legend + 256 glyphs) * 16 pixels/glyph */
00619 Height = 260 * 16; /* (2 header + 2 legend + 256 glyphs) * 16 rows/glyph */
00620
00621 ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */
00622
00623 FileSize = DataOffset + ImageSize;
00624
00625 /* convert dots/inch to pixels/meter */
00626 if (dpi == 0) dpi = 96;
00627 ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);
00628
00629 /*
00630  Generate the BMP Header
00631 */
00632 putchar ('B');
00633 putchar ('M');
00634 /*
00635  Calculate file size:
00636
00637  BMP Header + InfoHeader + Color Table + Raster Data
00638 */
00639 output4 (FileSize); /* FileSize */
00640 output4 (0x0000); /* reserved */
00641 /* Calculate DataOffset */
00642 output4 (DataOffset);
00643
00644 /*
00645  InfoHeader
00646 */
00647 output4 (40); /* Size of InfoHeader */
00648 output4 (Width); /* Width of bitmap in pixels */
00649 output4 (Height); /* Height of bitmap in pixels */
00650 output2 (1); /* Planes (1 plane) */
00651 output2 (1); /* BitCount (1 = monochrome) */
00652 output4 (0); /* Compression (0 = none) */
00653 output4 (ImageSize); /* ImageSize, in bytes */
00654 output4 (ppm); /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
00655 output4 (ppm); /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
00656 output4 (2); /* ColorsUsed (= 2) */
00657 output4 (2); /* ColorsImportant (= 2) */
00658 output4 (0x00000000); /* black (reserved, B, G, R) */
00659 output4 (0x00FFFFFF); /* white (reserved, B, G, R) */
00660
00661 /*
00662  Create header row bits.
00663 */
00664 snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
00665 memset ((void *)header, 0, 256 * 16 * sizeof (int)); /* fill with white */
00666 memset ((void *)header_string, ' ', 256 * sizeof (char)); /* 256 spaces */
00667 header_string[256] = '\0'; /* null-terminated */
00668

```



```

00669  hdrlen = strlen (raw_header);
00670  /* Wide bitmap can print 256 columns, but limit to 32 columns for long bitmap. */
00671  if (hdrlen > 32) hdrlen = 32;
00672  startcol = 127 - ((hdrlen - 1) » 1); /* to center header */
00673  /* center up to 32 chars */
00674  memcpy (&header_string[startcol], raw_header, hdrlen);
00675
00676  /* Copy each letter's bitmap from the plane_array[][] we constructed. */
00677  for (j = 0; j < 256; j++) {
00678      for (i = 0; i < 16; i++) {
00679          header[i][j] = ascii_bits[header_string[j] & 0x7F][i];
00680      }
00681  }
00682
00683  /*
00684   Create the left column legend.
00685  */
00686  memset ((void *)leftcol, 0, 256 * 16 * sizeof (unsigned));
00687
00688  for (codept = 0x0000; codept < 0x10000; codept += 0x100) {
00689      d1 = (codept » 12) & 0xF; /* most significant hex digit */
00690      d2 = (codept » 8) & 0xF;
00691
00692      thisrow = codept » 8; /* rows of 256 glyphs */
00693
00694      /* fill in first and second digits */
00695
00696      if (tinynum) { /* use 4x5 pixel glyphs */
00697          for (digitrow = 0; digitrow < 5; digitrow++) {
00698              leftcol[thisrow][6 + digitrow] =
00699                  (hexdigit[d1][digitrow] « 10) |
00700                  (hexdigit[d2][digitrow] « 4);
00701          }
00702      }
00703      else { /* bigger numbers -- use glyphs from Unifont itself */
00704          /* convert hexadecimal digits to ASCII equivalent */
00705          hexalpha1 = d1 < 0xA ? '0' + d1 : 'A' + d1 - 0xA;
00706          hexalpha2 = d2 < 0xA ? '0' + d2 : 'A' + d2 - 0xA;
00707
00708          for (i = 0; i < 16; i++) {
00709              leftcol[thisrow][i] =
00710                  (ascii_bits[hexalpha1][i] « 2) |
00711                  (ascii_bits[hexalpha2][i] » 6);
00712          }
00713      }
00714
00715      for (i = 0; i < 15; i++) {
00716          leftcol[thisrow][i] |= 0x00000002; /* right border */
00717      }
00718
00719      leftcol[thisrow][15] = 0x0000FFFE; /* bottom border */
00720
00721      if (d2 == 0xF) { /* 4096-point boundary */
00722          leftcol[thisrow][15] |= 0x00FF0000; /* longer tic mark */
00723      }
00724
00725      if ((thisrow % 0x40) == 0x3F) { /* 16,384-point boundary */
00726          leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
00727      }
00728  }
00729
00730  /*
00731   Create the top row legend.
00732  */
00733  memset ((void *)toprow, 0, 32 * 256 * sizeof (unsigned));
00734
00735  for (codept = 0x00; codept <= 0xFF; codept++) {
00736      d3 = (codept » 4) & 0xF;
00737      d4 = codept & 0xF; /* least significant hex digit */
00738
00739      if (tinynum) {
00740          for (digitrow = 0; digitrow < 5; digitrow++) {
00741              toprow[16 + 6 + digitrow][codept] =
00742                  (hexdigit[d3][digitrow] « 10) |
00743                  (hexdigit[d4][digitrow] « 4);
00744          }
00745      }
00746      else {
00747          /* convert hexadecimal digits to ASCII equivalent */
00748          hexalpha1 = d3 < 0xA ? '0' + d3 : 'A' + d3 - 0xA;
00749          hexalpha2 = d4 < 0xA ? '0' + d4 : 'A' + d4 - 0xA;

```

```

00750     for (i = 0 ; i < 16; i++) {
00751         toprow[14 + i][codept] =
00752             (ascii_bits[hexalpha1[i]    ) |
00753             (ascii_bits[hexalpha2[i] » 7]);
00754     }
00755 }
00756 }
00757
00758 for (j = 0; j < 256; j++) {
00759     /* force bottom pixel row to be white, for separation from glyphs */
00760     toprow[16 + 15][j] = 0x0000;
00761 }
00762
00763 /* 1 pixel row with left-hand legend line */
00764 for (j = 0; j < 256; j++) {
00765     toprow[16 + 14][j] |= 0xFFFF;
00766 }
00767
00768 /* 14 rows with line on left to fill out this character row */
00769 for (i = 13; i >= 0; i--) {
00770     for (j = 0; j < 256; j++) {
00771         toprow[16 + i][j] |= 0x0001;
00772     }
00773 }
00774
00775 /* Form the longer tic marks in top legend */
00776 for (i = 8; i < 16; i++) {
00777     for (j = 0x0F; j < 0x100; j += 0x10) {
00778         toprow[i][j] |= 0x0001;
00779     }
00780 }
00781
00782 /*
00783     Now write the raster image.
00784
00785     XOR each byte with 0xFF because black = 0, white = 1 in BMP.
00786 */
00787
00788 /* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
00789 for (i = 0xFF00; i >= 0; i -= 0x100) {
00790     thisrow = i » 8; /* 256 glyphs per row */
00791     for (j = 15; j >= 0; j--) {
00792         /* left-hand legend */
00793         putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
00794         putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
00795         putchar ((~leftcol[thisrow][j] » 8) & 0xFF);
00796         putchar ( ~leftcol[thisrow][j]      & 0xFF);
00797         /* Unifont glyph */
00798         for (k = 0x00; k < 0x100; k++) {
00799             bytesout = ~plane_array[i+k][j] & 0xFFFF;
00800             putchar ((bytesout » 8) & 0xFF);
00801             putchar ( bytesout      & 0xFF);
00802         }
00803     }
00804 }
00805
00806 /*
00807     Write the top legend.
00808 */
00809 /* i == 15: bottom pixel row of header is output here */
00810 /* left-hand legend: solid black line except for right-most pixel */
00811 putchar (0x00);
00812 putchar (0x00);
00813 putchar (0x00);
00814 putchar (0x01);
00815 for (j = 0; j < 256; j++) {
00816     putchar ((~toprow[16 + 15][j] » 8) & 0xFF);
00817     putchar ( ~toprow[16 + 15][j]      & 0xFF);
00818 }
00819
00820 putchar (0xFF);
00821 putchar (0xFF);
00822 putchar (0xFF);
00823 putchar (0xFC);
00824 for (j = 0; j < 256; j++) {
00825     putchar ((~toprow[16 + 14][j] » 8) & 0xFF);
00826     putchar ( ~toprow[16 + 14][j]      & 0xFF);
00827 }
00828
00829 for (i = 16 + 13; i >= 0; i--) {
00830     if (i >= 8) { /* make vertical stroke on right */

```

```

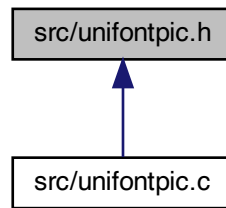
00831     putchar (0xFF);
00832     putchar (0xFF);
00833     putchar (0xFF);
00834     putchar (0xFD);
00835 }
00836 else { /* all white */
00837     putchar (0xFF);
00838     putchar (0xFF);
00839     putchar (0xFF);
00840     putchar (0xFF);
00841 }
00842 for (j = 0; j < 256; j++) {
00843     putchar ((~toprow[i][j] » 8) & 0xFF);
00844     putchar ( ~toprow[i][j]      & 0xFF);
00845 }
00846 }
00847
00848 /*
00849  * Write the header.
00850  */
00851
00852 /* 8 completely white rows */
00853 for (i = 7; i >= 0; i--) {
00854     for (j = 0; j < 258; j++) {
00855         putchar (0xFF);
00856         putchar (0xFF);
00857     }
00858 }
00859
00860 for (i = 15; i >= 0; i--) {
00861     /* left-hand legend */
00862     putchar (0xFF);
00863     putchar (0xFF);
00864     putchar (0xFF);
00865     putchar (0xFF);
00866     /* header glyph */
00867     for (j = 0; j < 256; j++) {
00868         bytesout = ~header[i][j] & 0xFFFF;
00869         putchar ((bytesout » 8) & 0xFF);
00870         putchar ( bytesout      & 0xFF);
00871     }
00872 }
00873
00874 /* 8 completely white rows at very top */
00875 for (i = 7; i >= 0; i--) {
00876     for (j = 0; j < 258; j++) {
00877         putchar (0xFF);
00878         putchar (0xFF);
00879     }
00880 }
00881
00882 return;
00883 }
00884

```

5.19 src/unifontpic.h File Reference

[unifontpic.h](#) - Header file for [unifontpic.c](#)

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



Macros

- `#define` [MAXSTRING](#) 256
Maximum input string allowed.
- `#define` [HEADER_STRING](#) "GNU Unifont 15.0.02"
To be printed as chart title.

Variables

- `const char *` [ascii_hex](#) [128]
Array of Unifont ASCII glyphs for chart row & column headings.
- `int` [ascii_bits](#) [128][16]
Array to hold ASCII bitmaps for chart title.
- `char` [hexdigit](#) [16][5]
Array of 4x5 hexadecimal digits for legend.

5.19.1 Detailed Description

[unifontpic.h](#) - Header file for [unifontpic.c](#)

Author

Paul Hardy, July 2017

Copyright

Copyright (C) 2017 Paul Hardy

Definition in file [unifontpic.h](#).

5.19.2 Macro Definition Documentation

5.19.2.1 HEADER_STRING

```
#define HEADER_STRING "GNU Unifont 15.0.02"
```

To be printed as chart title.

Definition at line 30 of file [unifontpic.h](#).

5.19.2.2 MAXSTRING

```
#define MAXSTRING 256
```

Maximum input string allowed.

Definition at line 28 of file [unifontpic.h](#).

5.19.3 Variable Documentation

5.19.3.1 ascii_bits

```
int ascii_bits[128][16]
```

Array to hold ASCII bitmaps for chart title.

This array will be created from the strings in `ascii_hex[]` above.

Definition at line 177 of file [unifontpic.h](#).

5.19.3.2 ascii_hex

```
const char* ascii_hex[128]
```

Array of Unifont ASCII glyphs for chart row & column headings.

Define the array of Unifont ASCII glyphs, code points 0 through 127. This allows using `unifontpic` to print charts of glyphs above Unicode Plane 0. These were copied from `font/plane00/unifont-base.hex`, plus U+0020 (ASCII space character).

Definition at line 40 of file [unifontpic.h](#).

5.19.3.3 hexdigit

```
char hexdigit[16][5]
```

Initial value:

```
= {
    {0x6,0x9,0x9,0x9,0x6},
    {0x2,0x6,0x2,0x2,0x7},
    {0xF,0x1,0xF,0x8,0xF},
    {0xE,0x1,0x7,0x1,0xE},
    {0x9,0x9,0xF,0x1,0x1},
    {0xF,0x8,0xF,0x1,0xF},
    {0x6,0x8,0xE,0x9,0x6},
    {0xF,0x1,0x2,0x4,0x4},
    {0x6,0x9,0x6,0x9,0x6},
    {0x6,0x9,0x7,0x1,0x6},
    {0xF,0x9,0xF,0x9,0x9},
    {0xE,0x9,0xE,0x9,0xE},
    {0x7,0x8,0x8,0x8,0x7},
    {0xE,0x9,0x9,0x9,0xE},
    {0xF,0x8,0xE,0x8,0xF},
    {0xF,0x8,0xE,0x8,0x8}
}
```

Array of 4x5 hexadecimal digits for legend.

hexdigit contains 4x5 pixel arrays of tiny digits for the legend. See [unihexgen.c](#) for a more detailed description in the comments.

Definition at line 186 of file [unifontpic.h](#).

5.20 unifontpic.h

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

```
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
00028 #define MAXSTRING 256 ///< Maximum input string allowed.
00029
00030 #define HEADER_STRING "GNU Unifont 15.0.02" ///< To be printed as chart title.
00031
00032 /**
00033  @brief Array of Unifont ASCII glyphs for chart row & column headings.
00034
00035  Define the array of Unifont ASCII glyphs, code points 0 through 127.
```

Generated by Doxygen

```

00117 "004C:0000000040404040404040407E0000",
00118 "004D:00000000424266665A5A424242420000",
00119 "004E:0000000042626252524A4A4646420000",
00120 "004F:000000003C424242424242423C0000",
00121 "0050:000000007C4242427C40404040400000",
00122 "0051:000000003C4242424242425A663C0300",
00123 "0052:000000007C4242427C48444442420000",
00124 "0053:000000003C424240300C0242423C0000",
00125 "0054:000000007F08080808080808080000",
00126 "0055:0000000042424242424242423C0000",
00127 "0056:00000000414141222222141408080000",
00128 "0057:00000000424242425A5A666642420000",
00129 "0058:000000004242424181824242420000",
00130 "0059:000000004141222214080808080000",
00131 "005A:000000007E02020408102040407E0000",
00132 "005B:00000000E0808080808080808080E00",
00133 "005C:00000000404020101008080402020000",
00134 "005D:00000070101010101010101010107000",
00135 "005E:00001824420000000000000000000000",
00136 "005F:00000000000000000000000000007F00",
00137 "0060:00201008000000000000000000000000",
00138 "0061:0000000000003C42023E4242463A0000",
00139 "0062:0000004040405C6242424242625C0000",
00140 "0063:0000000000003C4240404040423C0000",
00141 "0064:0000000202023A4642424242463A0000",
00142 "0065:0000000000003C42427E4040423C0000",
00143 "0066:0000000C1010107C1010101010100000",
00144 "0067:0000000000023A44444438203C42423C",
00145 "0068:0000004040405C624242424242420000",
00146 "0069:00000008080018080808080808083E0000",
00147 "006A:0000000404000C04040404040444830",
00148 "006B:00000040404044485060504844420000",
00149 "006C:00000018080808080808080808083E0000",
00150 "006D:00000000000076494949494949490000",
00151 "006E:0000000000005C624242424242420000",
00152 "006F:0000000000003C4242424242423C0000",
00153 "0070:0000000000005C6242424242625C4040",
00154 "0071:0000000000003A4642424242463A0202",
00155 "0072:0000000000005C624240404040400000",
00156 "0073:0000000000003C4240300C02423C0000",
00157 "0074:000000001010107C10101010100C0000",
00158 "0075:000000000000424242424242463A0000",
00159 "0076:0000000000004242424242418180000",
00160 "0077:00000000000041494949494949360000",
00161 "0078:00000000000042424181824242420000",
00162 "0079:0000000000004242424242261A02023C",
00163 "007A:0000000000007E0204081020407E0000",
00164 "007B:0000000C10100808102010080810100C",
00165 "007C:00000808080808080808080808080808",
00166 "007D:00000030080810100804081010080830",
00167 "007E:00000031494600000000000000000000",
00168 "007F:AAAA000180000001800073D1CA104BD1CA1073DF800000018000000180005555"
00169 };
00170
00171
00172 /**
00173  @brief Array to hold ASCII bitmaps for chart title.
00174
00175  This array will be created from the strings in ascii_hex[] above.
00176 */
00177 int ascii_bits[128][16];
00178
00179
00180 /**
00181  @brief Array of 4x5 hexadecimal digits for legend.
00182
00183  hexdigit contains 4x5 pixel arrays of tiny digits for the legend.
00184  See unihexgen.c for a more detailed description in the comments.
00185 */
00186 char hexdigit[16][5] = {
00187  {0x6,0x9,0x9,0x9,0x6}, /* 0x0 */
00188  {0x2,0x6,0x2,0x2,0x7}, /* 0x1 */
00189  {0xF,0x1,0xF,0x8,0xF}, /* 0x2 */
00190  {0xE,0x1,0x7,0x1,0xE}, /* 0x3 */
00191  {0x9,0x9,0xF,0x1,0x1}, /* 0x4 */
00192  {0xF,0x8,0xF,0x1,0xF}, /* 0x5 */
00193  {0x6,0x8,0xE,0x9,0x6}, /* 0x6 */
00194  {0xF,0x1,0x2,0x4,0x4}, /* 0x7 */
00195  {0x6,0x9,0x6,0x9,0x6}, /* 0x8 */
00196  {0x6,0x9,0x7,0x1,0x6}, /* 0x9 */
00197  {0xF,0x9,0xF,0x9,0x9}, /* 0xA */

```



```

00198  {0xE,0x9,0xE,0x9,0xE}, /* 0xB */
00199  {0x7,0x8,0x8,0x8,0x7}, /* 0xC */
00200  {0xE,0x9,0x9,0x9,0xE}, /* 0xD */
00201  {0xF,0x8,0xE,0x8,0xF}, /* 0xE */
00202  {0xF,0x8,0xE,0x8,0x8} /* 0xF */
00203  };
00204

```

5.21 src/unigencircles.c File Reference

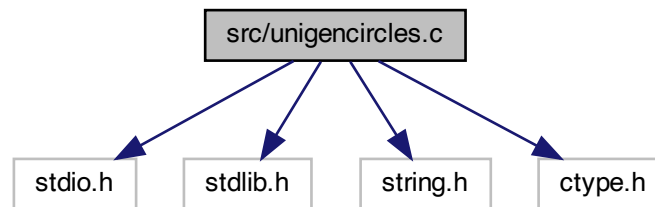
unigencircles - Superimpose dashed combining circles on combining glyphs

```

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

```

Include dependency graph for unigencircles.c:



Macros

- `#define MAXSTRING 256`
Maximum input line length - 1.

Functions

- `int main (int argc, char **argv)`
The main function.
- `void add_single_circle (char *glyphstring)`
Superimpose a single-width dashed combining circle on a glyph bitmap.
- `void add_double_circle (char *glyphstring, int offset)`
Superimpose a double-width dashed combining circle on a glyph bitmap.

5.21.1 Detailed Description

unigencircles - Superimpose dashed combining circles on combining glyphs

Author

Paul Hardy

Copyright

Copyright (C) 2013, Paul Hardy.

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

5.21.2 Macro Definition Documentation

5.21.2.1 MAXSTRING

```
#define MAXSTRING 256
```

Maximum input line length - 1.

Definition at line [62](#) of file [unigencircles.c](#).

5.21.3 Function Documentation

5.21.3.1 add_double_circle()

```
void add_double_circle (  
    char * glyphstring,  
    int offset )
```

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

Parameters

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

Definition at line 221 of file [unigencircles.c](#).

```

00222 {
00223
00224     char newstring[256];
00225     /* Circle hex string pattern is "00000008000024004200240000000000" */
00226
00227     /* For double diacritical glyphs (offset = -8) */
00228     /* Combining circle is left-justified. */
00229     char circle08[64]={0x0,0x0,0x0,0x0, /* row 1 */
00230                        0x0,0x0,0x0,0x0, /* row 2 */
00231                        0x0,0x0,0x0,0x0, /* row 3 */
00232                        0x0,0x0,0x0,0x0, /* row 4 */
00233                        0x0,0x0,0x0,0x0, /* row 5 */
00234                        0x0,0x0,0x0,0x0, /* row 6 */
00235                        0x2,0x4,0x0,0x0, /* row 7 */
00236                        0x0,0x0,0x0,0x0, /* row 8 */
00237                        0x4,0x2,0x0,0x0, /* row 9 */
00238                        0x0,0x0,0x0,0x0, /* row 10 */
00239                        0x2,0x4,0x0,0x0, /* row 11 */
00240                        0x0,0x0,0x0,0x0, /* row 12 */
00241                        0x0,0x0,0x0,0x0, /* row 13 */
00242                        0x0,0x0,0x0,0x0, /* row 14 */
00243                        0x0,0x0,0x0,0x0, /* row 15 */
00244                        0x0,0x0,0x0,0x0}; /* row 16 */
00245
00246     /* For all other combining glyphs (offset = -16) */
00247     /* Combining circle is centered in 16 columns. */
00248     char circle16[64]={0x0,0x0,0x0,0x0, /* row 1 */
00249                        0x0,0x0,0x0,0x0, /* row 2 */
00250                        0x0,0x0,0x0,0x0, /* row 3 */
00251                        0x0,0x0,0x0,0x0, /* row 4 */
00252                        0x0,0x0,0x0,0x0, /* row 5 */
00253                        0x0,0x0,0x0,0x0, /* row 6 */
00254                        0x0,0x2,0x4,0x0, /* row 7 */
00255                        0x0,0x0,0x0,0x0, /* row 8 */
00256                        0x0,0x4,0x2,0x0, /* row 9 */
00257                        0x0,0x0,0x0,0x0, /* row 10 */
00258                        0x0,0x2,0x4,0x0, /* row 11 */
00259                        0x0,0x0,0x0,0x0, /* row 12 */
00260                        0x0,0x0,0x0,0x0, /* row 13 */
00261                        0x0,0x0,0x0,0x0, /* row 14 */
00262                        0x0,0x0,0x0,0x0, /* row 15 */
00263                        0x0,0x0,0x0,0x0}; /* row 16 */
00264
00265     char *circle; /* points into circle16 or circle08 */
00266
00267     int digit1, digit2; /* corresponding digits in each string */
00268
00269     int i; /* index variables */
00270
00271
00272     /*
00273      Determine if combining circle is left-justified (offset = -8)
00274      or centered (offset = -16).
00275     */
00276     circle = (offset >= -8) ? circle08 : circle16;
00277
00278     /* for each character position, OR the corresponding circle glyph value */
00279     for (i = 0; i < 64; i++) {
00280         glyphstring[i] = toupper (glyphstring[i]);
00281
00282         /* Convert ASCII character to a hexadecimal integer */
00283         digit1 = (glyphstring[i] <= '9') ?
00284                 (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);
00285
00286         /* Superimpose dashed circle */

```

```

00287     digit2 = digit1 | circle[i];
00288
00289     /* Convert hexadecimal integer to an ASCII character */
00290     newstring[i] = (digit2 <= 9) ?
00291         ('0' + digit2) : ('A' + digit2 - 0xA);
00292 }
00293
00294 /* Terminate string for output */
00295 newstring[i++] = '\n';
00296 newstring[i++] = '\0';
00297
00298 memcpy (glyphstring, newstring, i);
00299
00300 return;
00301 }

```

Here is the caller graph for this function:



5.21.3.2 add_single_circle()

```

void add_single_circle (
    char * glyphstring )

```

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

Parameters

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

Definition at line 163 of file [unigencircles.c](#).

```

00164 {
00165
00166     char newstring[256];
00167     /* Circle hex string pattern is "00000008000024004200240000000000" */
00168     char circle[32]={0x0,0x0, /* row 1 */
00169         0x0,0x0, /* row 2 */
00170         0x0,0x0, /* row 3 */
00171         0x0,0x0, /* row 4 */
00172         0x0,0x0, /* row 5 */
00173         0x0,0x0, /* row 6 */
00174         0x2,0x4, /* row 7 */

```

```

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

```

Here is the caller graph for this function:



5.21.3.3 main()

```

int main (
    int argc,
    char ** argv )

```

The main function.

Parameters

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

Returns

This program exits with status EXIT_SUCCESS.

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

```

00074 {
00075
00076 char teststring[MAXSTRING]; /* current input line */
00077 int loc; /* Unicode code point of current input line */
00078 int offset; /* offset value of a combining character */
00079 char *gstart; /* glyph start, pointing into teststring */
00080
00081 char combining[0x110000]; /* 1 --> combining glyph; 0 --> non-combining */
00082 char x_offset [0x110000]; /* second value in *combining.txt files */
00083
00084 void add_single_circle(char *); /* add a single-width dashed circle */
00085 void add_double_circle(char *, int); /* add a double-width dashed circle */
00086
00087 FILE *infilefp;
00088
00089 /*
00090  if (argc != 3) {
00091      fprintf(stderr,
00092          "\n\nUsage: %s combining.txt nonprinting.hex < unifont.hex > unifontfull.hex\n\n");
00093      exit (EXIT_FAILURE);
00094  }
00095  */
00096
00097 /*
00098  Read the combining characters list.
00099  */
00100 /* Start with no combining code points flagged */
00101 memset (combining, 0, 0x110000 * sizeof (char));
00102 memset (x_offset , 0, 0x110000 * sizeof (char));
00103
00104 if ((infilefp = fopen (argv[1], "r")) == NULL) {
00105     fprintf (stderr, "ERROR - combining characters file %s not found.\n\n",
00106         argv[1]);
00107     exit (EXIT_FAILURE);
00108 }
00109
00110 /* Flag list of combining characters to add a dashed circle. */
00111 while (fscanf (infilefp, "%X:%d", &loc, &offset) != EOF) {
00112     /*
00113      U+01107F and U+01D1A0 are not defined as combining characters
00114      in Unicode; they were added in a combining.txt file as the
00115      only way to make them look acceptable in proximity to other
00116      glyphs in their script.

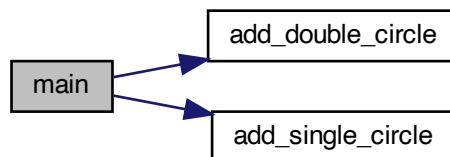
```

```

00117     */
00118     if (loc != 0x01107F && loc != 0x01D1A0) {
00119         combining[loc] = 1;
00120         x_offset[loc] = offset;
00121     }
00122 }
00123 fclose (infilep); /* all done reading combining.txt */
00124
00125 /* Now read the non-printing glyphs; they never have dashed circles */
00126 if ((infilep = fopen (argv[2], "r")) == NULL) {
00127     fprintf (stderr, "ERROR - nonprinting characters file %s not found.\n\n",
00128             argv[1]);
00129     exit (EXIT_FAILURE);
00130 }
00131
00132 /* Reset list of nonprinting characters to avoid adding a dashed circle. */
00133 while (fscanf (infilep, "%X:%*s", &loc) != EOF) combining[loc] = 0;
00134
00135 fclose (infilep); /* all done reading nonprinting.hex */
00136
00137 /*
00138  * Read the hex glyphs.
00139  */
00140 teststring[MAXSTRING - 1] = '\0'; /* so there's no chance we leave array */
00141 while (fgets (teststring, MAXSTRING-1, stdin) != NULL) {
00142     sscanf (teststring, "%X", &loc); /* loc == the Unicode code point */
00143     gstart = strchr (teststring, ':') + 1; /* start of glyph bitmap */
00144     if (combining[loc]) { /* if a combining character */
00145         if (strlen (gstart) < 35)
00146             add_single_circle (gstart); /* single-width */
00147         else
00148             add_double_circle (gstart, x_offset[loc]); /* double-width */
00149     }
00150     printf ("%s", teststring); /* output the new character .hex string */
00151 }
00152
00153 exit (EXIT_SUCCESS);
00154 }

```

Here is the call graph for this function:



5.22 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
00056 #include <stdio.h>
00057 #include <stdlib.h>
00058 #include <string.h>
00059 #include <ctype.h>
00060
00061 #define MAXSTRING 256 ///< Maximum input line length - 1.
00062
00063
00064 /**
00065 @brief The main function.
00066
00067 @param[in] argc The count of command line arguments.
00068 @param[in] argv Pointer to array of command line arguments.
00069 @return This program exits with status EXIT_SUCCESS.
00070 */
00071
00072 int
00073 main (int argc, char **argv)
00074 {
00075     char teststring[MAXSTRING]; /* current input line */
00076     int loc; /* Unicode code point of current input line */
00077     int offset; /* offset value of a combining character */
00078     char *gstart; /* glyph start, pointing into teststring */
00079
00080     char combining[0x110000]; /* 1 --> combining glyph; 0 --> non-combining */
00081     char x_offset [0x110000]; /* second value in combining.txt files */
00082
00083     void add_single_circle(char *); /* add a single-width dashed circle */
00084     void add_double_circle(char *, int); /* add a double-width dashed circle */
00085
00086     FILE *infilefp;
00087
00088     /*
00089     if (argc != 3) {
00090         fprintf (stderr,

```



```

00092         "\n\nUsage: %s combining.txt nonprinting.hex < unifont.hex > unifontfull.hex\n\n");
00093     exit (EXIT_FAILURE);
00094 }
00095 */
00096
00097 /*
00098  Read the combining characters list.
00099 */
00100 /* Start with no combining code points flagged */
00101 memset (combining, 0, 0x110000 * sizeof (char));
00102 memset (x_offset, 0, 0x110000 * sizeof (char));
00103
00104 if ((infilep = fopen (argv[1], "r")) == NULL) {
00105     fprintf (stderr, "ERROR - combining characters file %s not found.\n\n",
00106             argv[1]);
00107     exit (EXIT_FAILURE);
00108 }
00109
00110 /* Flag list of combining characters to add a dashed circle. */
00111 while (fscanf (infilep, "%X:%d", &loc, &offset) != EOF) {
00112     /*
00113      U+01107F and U+01D1A0 are not defined as combining characters
00114      in Unicode; they were added in a combining.txt file as the
00115      only way to make them look acceptable in proximity to other
00116      glyphs in their script.
00117     */
00118     if (loc != 0x01107F && loc != 0x01D1A0) {
00119         combining[loc] = 1;
00120         x_offset[loc] = offset;
00121     }
00122 }
00123 fclose (infilep); /* all done reading combining.txt */
00124
00125 /* Now read the non-printing glyphs; they never have dashed circles */
00126 if ((infilep = fopen (argv[2], "r")) == NULL) {
00127     fprintf (stderr, "ERROR - nonprinting characters file %s not found.\n\n",
00128             argv[1]);
00129     exit (EXIT_FAILURE);
00130 }
00131
00132 /* Reset list of nonprinting characters to avoid adding a dashed circle. */
00133 while (fscanf (infilep, "%X:%s", &loc) != EOF) combining[loc] = 0;
00134
00135 fclose (infilep); /* all done reading nonprinting.hex */
00136
00137 /*
00138  Read the hex glyphs.
00139 */
00140 teststring[MAXSTRING - 1] = '\0'; /* so there's no chance we leave array */
00141 while (fgets (teststring, MAXSTRING-1, stdin) != NULL) {
00142     sscanf (teststring, "%X", &loc); /* loc == the Unicode code point */
00143     gstart = strchr (teststring, ':') + 1; /* start of glyph bitmap */
00144     if (combining[loc]) { /* if a combining character */
00145         if (strlen (gstart) < 35)
00146             add_single_circle (gstart); /* single-width */
00147         else
00148             add_double_circle (gstart, x_offset[loc]); /* double-width */
00149     }
00150     printf ("%s", teststring); /* output the new character .hex string */
00151 }
00152
00153 exit (EXIT_SUCCESS);
00154 }
00155
00156 /**
00157  @brief Superimpose a single-width dashed combining circle on a glyph bitmap.
00158
00159  @param[in,out] glyphstring A single-width glyph, 8x16 pixels.
00160 */
00161 void
00162 add_single_circle (char *glyphstring)
00163 {
00164     char newstring[256];
00165     /* Circle hex string pattern is "000000008000024004200240000000000" */
00166     char circle[32] = {0x0, 0x0, /* row 1 */
00167                       0x0, 0x0, /* row 2 */
00168                       0x0, 0x0, /* row 3 */
00169                       0x0, 0x0, /* row 4 */
00170                       0x0, 0x0, /* row 5 */

```

```

00173         0x0,0x0, /* row 6 */
00174         0x2,0x4, /* row 7 */
00175         0x0,0x0, /* row 8 */
00176         0x4,0x2, /* row 9 */
00177         0x0,0x0, /* row 10 */
00178         0x2,0x4, /* row 11 */
00179         0x0,0x0, /* row 12 */
00180         0x0,0x0, /* row 13 */
00181         0x0,0x0, /* row 14 */
00182         0x0,0x0, /* row 15 */
00183         0x0,0x0}; /* row 16 */
00184
00185 int digit1, digit2; /* corresponding digits in each string */
00186
00187 int i; /* index variables */
00188
00189 /* for each character position, OR the corresponding circle glyph value */
00190 for (i = 0; i < 32; i++) {
00191     glyphstring[i] = toupper (glyphstring[i]);
00192
00193     /* Convert ASCII character to a hexadecimal integer */
00194     digit1 = (glyphstring[i] <= '9') ?
00195         (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);
00196
00197     /* Superimpose dashed circle */
00198     digit2 = digit1 | circle[i];
00199
00200     /* Convert hexadecimal integer to an ASCII character */
00201     newstring[i] = (digit2 <= 9) ?
00202         ('0' + digit2) : ('A' + digit2 - 0xA);
00203 }
00204
00205 /* Terminate string for output */
00206 newstring[i++] = '\n';
00207 newstring[i++] = '\0';
00208
00209 memcpy (glyphstring, newstring, i);
00210
00211 return;
00212 }
00213
00214
00215 /**
00216  @brief Superimpose a double-width dashed combining circle on a glyph bitmap.
00217
00218  @param[in,out] glyphstring A double-width glyph, 16x16 pixels.
00219 */
00220 void
00221 add_double_circle (char *glyphstring, int offset)
00222 {
00223
00224     char newstring[256];
00225     /* Circle hex string pattern is "00000008000024004200240000000000" */
00226
00227     /* For double diacritical glyphs (offset = -8) */
00228     /* Combining circle is left-justified. */
00229     char circle08[64]={0x0,0x0,0x0,0x0, /* row 1 */
00230         0x0,0x0,0x0,0x0, /* row 2 */
00231         0x0,0x0,0x0,0x0, /* row 3 */
00232         0x0,0x0,0x0,0x0, /* row 4 */
00233         0x0,0x0,0x0,0x0, /* row 5 */
00234         0x0,0x0,0x0,0x0, /* row 6 */
00235         0x2,0x4,0x0,0x0, /* row 7 */
00236         0x0,0x0,0x0,0x0, /* row 8 */
00237         0x4,0x2,0x0,0x0, /* row 9 */
00238         0x0,0x0,0x0,0x0, /* row 10 */
00239         0x2,0x4,0x0,0x0, /* row 11 */
00240         0x0,0x0,0x0,0x0, /* row 12 */
00241         0x0,0x0,0x0,0x0, /* row 13 */
00242         0x0,0x0,0x0,0x0, /* row 14 */
00243         0x0,0x0,0x0,0x0, /* row 15 */
00244         0x0,0x0,0x0,0x0}; /* row 16 */
00245
00246     /* For all other combining glyphs (offset = -16) */
00247     /* Combining circle is centered in 16 columns. */
00248     char circle16[64]={0x0,0x0,0x0,0x0, /* row 1 */
00249         0x0,0x0,0x0,0x0, /* row 2 */
00250         0x0,0x0,0x0,0x0, /* row 3 */
00251         0x0,0x0,0x0,0x0, /* row 4 */
00252         0x0,0x0,0x0,0x0, /* row 5 */
00253         0x0,0x0,0x0,0x0, /* row 6 */

```

```

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

```

5.23 src/unigenwidth.c File Reference

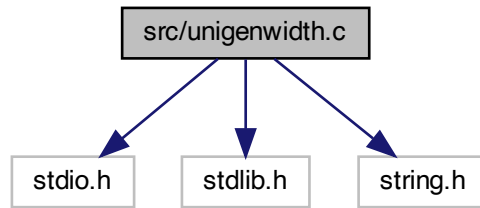
unigenwidth - IEEE 1003.1-2008 setup to calculate wchar_t string widths

```

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

```

Include dependency graph for unigenwidth.c:



Macros

- `#define MAXSTRING 256`
Maximum input line length - 1.
- `#define PIKTO_START 0x0F0E70`
Start of Pikto code point range.
- `#define PIKTO_END 0x0F11EF`
End of Pikto code point range.
- `#define PIKTO_SIZE (PIKTO_END - PIKTO_START + 1)`

Functions

- `int main (int argc, char **argv)`
The main function.

5.23.1 Detailed Description

unigenwidth - IEEE 1003.1-2008 setup to calculate `wchar_t` string widths

Author

Paul Hardy.

Copyright

Copyright (C) 2013, 2017 Paul Hardy.

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

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

5.23.2 Macro Definition Documentation

5.23.2.1 MAXSTRING

```
#define MAXSTRING 256
```

Maximum input line length - 1.

Definition at line 46 of file [unigenwidth.c](#).

5.23.2.2 PIKTO__END

```
#define PIKTO__END 0xF11EF
```

End of Pikto code point range.

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

5.23.2.3 PIKTO__SIZE

```
#define PIKTO__SIZE (PIKTO__END - PIKTO__START + 1)
```

Number of code points in Pikto range.

Definition at line 52 of file [unigenwidth.c](#).

5.23.2.4 PIKTO__START

```
#define PIKTO__START 0xF0E70
```

Start of Pikto code point range.

Definition at line 49 of file [unigenwidth.c](#).

5.23.3 Function Documentation

5.23.3.1 main()

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

The main function.

Parameters

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

Returns

This program exits with status `EXIT_SUCCESS`.

Definition at line 63 of file `unigenwidth.c`.

```

00064 {
00065
00066     int i; /* loop variable */
00067
00068     char teststring[MAXSTRING];
00069     int loc;
00070     char *gstart;
00071
00072     char glyph_width[0x20000];
00073     char pikto_width[PIKTO_SIZE];
00074
00075     FILE *infilefp;
00076
00077     if (argc != 3) {
00078         fprintf(stderr, "\n\nUsage: %s <unifont.hex> <combining.txt>\n\n", argv[0]);
00079         exit (EXIT_FAILURE);
00080     }
00081
00082     /*
00083      * Read the collection of hex glyphs.
00084      */
00085     if ((infilefp = fopen (argv[1], "r")) == NULL) {
00086         fprintf(stderr, "ERROR - hex input file %s not found.\n\n", argv[1]);
00087         exit (EXIT_FAILURE);
00088     }
00089
00090     /* Flag glyph as non-existent until found. */
00091     memset (glyph_width, -1, 0x20000 * sizeof (char));
00092     memset (pikto_width, -1, (PIKTO_SIZE) * sizeof (char));
00093
00094     teststring[MAXSTRING-1] = '\0';
00095     while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
00096         sscanf (teststring, "%X:%s", &loc);
00097         if (loc < 0x20000) {
00098             gstart = strchr (teststring, ':') + 1;
00099             /*
00100              * 16 rows per glyph, 2 ASCII hexadecimal digits per byte,
00101              * so divide number of digits by 32 (shift right 5 bits).
00102              */
00103             glyph_width[loc] = (strlen (gstart) - 1) » 5;
00104         }
00105         else if ((loc >= PIKTO_START) && (loc <= PIKTO_END)) {
00106             gstart = strchr (teststring, ':') + 1;

```

```

00107     pikto_width[loc - PIKTO_START] = strlen (gstart) <= 34 ? 1 : 2;
00108     }
00109 }
00110
00111 fclose (infilefp);
00112
00113 /*
00114  Now read the combining character code points.  These have width of 0.
00115 */
00116 if ((infilefp = fopen (argv[2], "r")) == NULL) {
00117     fprintf (stderr, "ERROR - combining characters file %s not found.\n\n", argv[2]);
00118     exit (EXIT_FAILURE);
00119 }
00120
00121 while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
00122     sscanf (teststring, "%X:%*s", &loc);
00123     if (loc < 0x20000) glyph_width[loc] = 0;
00124 }
00125
00126 fclose (infilefp);
00127
00128 /*
00129  Code Points with Unusual Properties (Unicode Standard, Chapter 4).
00130
00131  As of Unifont 10.0.04, use the widths in the *-nonprinting.hex"
00132  files.  If an application is smart enough to know how to handle
00133  these special cases, it will not render the "nonprinting" glyph
00134  and will treat the code point as being zero-width.
00135 */
00136 // glyph_width[0]=0; /* NULL character */
00137 // for (i = 0x0001; i <= 0x001F; i++) glyph_width[i]=-1; /* Control Characters */
00138 // for (i = 0x007F; i <= 0x009F; i++) glyph_width[i]=-1; /* Control Characters */
00139
00140 // glyph_width[0x034F]=0; /* combining grapheme joiner */
00141 // glyph_width[0x180B]=0; /* Mongolian free variation selector one */
00142 // glyph_width[0x180C]=0; /* Mongolian free variation selector two */
00143 // glyph_width[0x180D]=0; /* Mongolian free variation selector three */
00144 // glyph_width[0x180E]=0; /* Mongolian vowel separator */
00145 // glyph_width[0x200B]=0; /* zero width space */
00146 // glyph_width[0x200C]=0; /* zero width non-joiner */
00147 // glyph_width[0x200D]=0; /* zero width joiner */
00148 // glyph_width[0x200E]=0; /* left-to-right mark */
00149 // glyph_width[0x200F]=0; /* right-to-left mark */
00150 // glyph_width[0x202A]=0; /* left-to-right embedding */
00151 // glyph_width[0x202B]=0; /* right-to-left embedding */
00152 // glyph_width[0x202C]=0; /* pop directional formatting */
00153 // glyph_width[0x202D]=0; /* left-to-right override */
00154 // glyph_width[0x202E]=0; /* right-to-left override */
00155 // glyph_width[0x2060]=0; /* word joiner */
00156 // glyph_width[0x2061]=0; /* function application */
00157 // glyph_width[0x2062]=0; /* invisible times */
00158 // glyph_width[0x2063]=0; /* invisible separator */
00159 // glyph_width[0x2064]=0; /* invisible plus */
00160 // glyph_width[0x206A]=0; /* inhibit symmetric swapping */
00161 // glyph_width[0x206B]=0; /* activate symmetric swapping */
00162 // glyph_width[0x206C]=0; /* inhibit arabic form shaping */
00163 // glyph_width[0x206D]=0; /* activate arabic form shaping */
00164 // glyph_width[0x206E]=0; /* national digit shapes */
00165 // glyph_width[0x206F]=0; /* nominal digit shapes */
00166
00167 // /* Variation Selector-1 to Variation Selector-16 */
00168 // for (i = 0xFE00; i <= 0xFE0F; i++) glyph_width[i] = 0;
00169
00170 // glyph_width[0xFEFF]=0; /* zero width no-break space */
00171 // glyph_width[0xFFFF9]=0; /* interlinear annotation anchor */
00172 // glyph_width[0xFFFFA]=0; /* interlinear annotation separator */
00173 // glyph_width[0xFFFFB]=0; /* interlinear annotation terminator */
00174 /*
00175  Let glyph widths represent 0xFFFC (object replacement character)
00176  and 0xFFFD (replacement character).
00177 */
00178
00179 /*
00180  Hangul Jamo:
00181
00182  Leading Consonant (Choseong): leave spacing as is.
00183
00184  Hangul Choseong Filler (U+115F): set width to 2.
00185
00186  Hangul Jungseong Filler, Hangul Vowel (Jungseong), and
00187  Final Consonant (Jongseong): set width to 0, because these

```

```

00188         combine with the leading consonant as one composite syllabic
00189         glyph. As of Unicode 5.2, the Hangul Jamo block (U+1100..U+11FF)
00190         is completely filled.
00191     */
00192     // for (i = 0x1160; i <= 0x11FF; i++) glyph_width[i]=0; /* Vowels & Final Consonants */
00193
00194     /*
00195     Private Use Area -- the width is undefined, but likely
00196     to be 2 charcells wide either from a graphic glyph or
00197     from a four-digit hexadecimal glyph representing the
00198     code point. Therefore if any PUA glyph does not have
00199     a non-zero width yet, assign it a default width of 2.
00200     The Unicode Standard allows giving PUA characters
00201     default property values; see for example The Unicode
00202     Standard Version 5.0, p. 91. This same default is
00203     used for higher plane PUA code points below.
00204     */
00205     // for (i = 0xE000; i <= 0xF8FF; i++) {
00206     //     if (glyph_width[i] == 0) glyph_width[i]=2;
00207     // }
00208
00209     /*
00210     <not a character>
00211     */
00212     for (i = 0xFDD0; i <= 0xFDEF; i++) glyph_width[i] = -1;
00213     glyph_width[0xFFFFE] = -1; /* Byte Order Mark */
00214     glyph_width[0xFFFFF] = -1; /* Byte Order Mark */
00215
00216     /* Surrogate Code Points */
00217     for (i = 0xD800; i <= 0xDFFF; i++) glyph_width[i]=-1;
00218
00219     /* CJK Code Points */
00220     for (i = 0x4E00; i <= 0x9FFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00221     for (i = 0x3400; i <= 0x4DBF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00222     for (i = 0xF900; i <= 0xFAFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00223
00224     /*
00225     Now generate the output file.
00226     */
00227     printf ("%s\n");
00228     printf (" wewidth and wswidth functions, as per IEEE 1003.1-2008\n");
00229     printf (" System Interfaces, pp. 2241 and 2251.\n");
00230     printf (" Author: Paul Hardy, 2013\n");
00231     printf (" Copyright (c) 2013 Paul Hardy\n");
00232     printf (" LICENSE:\n");
00233     printf ("\n");
00234     printf (" This program is free software: you can redistribute it and/or modify\n");
00235     printf (" it under the terms of the GNU General Public License as published by\n");
00236     printf (" the Free Software Foundation, either version 2 of the License, or\n");
00237     printf (" (at your option) any later version.\n");
00238     printf ("\n");
00239     printf (" This program is distributed in the hope that it will be useful,\n");
00240     printf (" but WITHOUT ANY WARRANTY; without even the implied warranty of\n");
00241     printf (" MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the\n");
00242     printf (" GNU General Public License for more details.\n");
00243     printf ("\n");
00244     printf (" You should have received a copy of the GNU General Public License\n");
00245     printf (" along with this program. If not, see <http://www.gnu.org/licenses/>.\n");
00246     printf ("*/\n");
00247
00248     printf ("#include <wchar.h>\n");
00249     printf ("/* Definitions for Picto CSUR Private Use Area glyphs */\n");
00250     printf ("#define PIKTO_START\t0x%06X\n", PIKTO_START);
00251     printf ("#define PIKTO_END\t0x%06X\n", PIKTO_END);
00252     printf ("#define PIKTO_SIZE\t(PIKTO_END - PIKTO_START + 1)\n");
00253     printf ("\n");
00254     printf ("/* wewidth -- return charcell positions of one code point */\n");
00255     printf ("inline int nwewidth (wchar_t wc)\n");
00256     printf ("return (wswidth (&wc, 1));\n");
00257     printf ("}\n");
00258     printf ("\n");
00259     printf ("int nwswidth (const wchar_t *pwcs, size_t n)\n");
00260     printf ("int i; /* loop variable */\n");
00261     printf ("unsigned codept; /* Unicode code point of current character */\n");
00262     printf ("unsigned plane; /* Unicode plane, 0x00..0x10 */\n");
00263     printf ("unsigned lower17; /* lower 17 bits of Unicode code point */\n");
00264     printf ("unsigned lower16; /* lower 16 bits of Unicode code point */\n");
00265     printf ("int lowpt, midpt, highpt; /* for binary searching in plane zeroes[] */\n");
00266     printf ("int found; /* for binary searching in plane zeroes[] */\n");
00267     printf ("int totalwidth; /* total width of string, in charcells (1 or 2/glyph) */\n");
00268     printf ("int illegalchar; /* Whether or not this code point is illegal */\n");

```



```

00269 putchar ('\n');
00270
00271 /*
00272  Print the glyph_width[] array for glyphs widths in the
00273  Basic Multilingual Plane (Plane 0).
00274 */
00275 printf (" char glyph_width[0x20000] = {");
00276 for (i = 0; i < 0x10000; i++) {
00277     if ((i & 0x1F) == 0)
00278         printf ("\n /* U+%04X */ ", i);
00279     printf ("%d", glyph_width[i]);
00280 }
00281 for (i = 0x10000; i < 0x20000; i++) {
00282     if ((i & 0x1F) == 0)
00283         printf ("\n /* U+%06X */ ", i);
00284     printf ("%d", glyph_width[i]);
00285     if (i < 0x1FFFFF) putchar (',' );
00286 }
00287 printf ("\n }; \n\n");
00288
00289 /*
00290  Print the pikto_width[] array for Pikto glyph widths.
00291 */
00292 printf (" char pikto_width[PIKTO_SIZE] = {");
00293 for (i = 0; i < PIKTO_SIZE; i++) {
00294     if ((i & 0x1F) == 0)
00295         printf ("\n /* U+%06X */ ", PIKTO_START + i);
00296     printf ("%d", pikto_width[i]);
00297     if ((PIKTO_START + i) < PIKTO_END) putchar (',' );
00298 }
00299 printf ("\n }; \n\n");
00300
00301 /*
00302  Execution part of wcswidth.
00303 */
00304 printf ("\n");
00305 printf (" illegalchar = totalwidth = 0; \n");
00306 printf (" for (i = 0; !illegalchar && i < n; i++) { \n");
00307 printf ("     codept = pwc[i]; \n");
00308 printf ("     plane = codept » 16; \n");
00309 printf ("     lower17 = codept & 0x1FFFFF; \n");
00310 printf ("     lower16 = codept & 0xFFFF; \n");
00311 printf ("     if (plane < 2) { /* the most common case */ \n");
00312 printf ("         if (glyph_width[lower17] < 0) illegalchar = 1; \n");
00313 printf ("         else totalwidth += glyph_width[lower17]; \n");
00314 printf ("     } \n");
00315 printf ("     else { /* a higher plane or beyond Unicode range */ \n");
00316 printf ("         if ((lower16 == 0xFFFE) || (lower16 == 0xFFFF)) { \n");
00317 printf ("             illegalchar = 1; \n");
00318 printf ("         } \n");
00319 printf ("         else if (plane < 4) { /* Ideographic Plane */ \n");
00320 printf ("             totalwidth += 2; /* Default ideographic width */ \n");
00321 printf ("         } \n");
00322 printf ("         else if (plane == 0x0F) { /* CSUR Private Use Area */ \n");
00323 printf ("             if (lower16 <= 0x0E6F) { /* Kinya */ \n");
00324 printf ("                 totalwidth++; /* all Kinya syllables have width 1 */ \n");
00325 printf ("             } \n");
00326 printf ("             else if (lower16 <= (PIKTO_END & 0xFFFF)) { /* Pikto */ \n");
00327 printf ("                 if (pikto_width[lower16 - (PIKTO_START & 0xFFFF)] < 0) illegalchar = 1; \n");
00328 printf ("                 else totalwidth += pikto_width[lower16 - (PIKTO_START & 0xFFFF)]; \n");
00329 printf ("             } \n");
00330 printf ("         } \n");
00331 printf ("         else if (plane > 0x10) { \n");
00332 printf ("             illegalchar = 1; \n");
00333 printf ("         } \n");
00334 printf ("         /* Other non-printing in higher planes; return -1 as per IEEE 1003.1-2008. */ \n");
00335 printf ("         else if (/* language tags */ \n");
00336 printf ("             codept == 0x0E0001 || (codept >= 0x0E0020 && codept <= 0x0E007F) || \n");
00337 printf ("             /* variation selectors, 0x0E0100..0x0E01EF */ \n");
00338 printf ("             (codept >= 0x0E0100 && codept <= 0x0E01EF)) { \n");
00339 printf ("                 illegalchar = 1; \n");
00340 printf ("             } \n");
00341 printf ("         /* \n");
00342 printf ("         Unicode plane 0x02..0x10 printing character \n");
00343 printf ("         */ \n");
00344 printf ("         else { \n");
00345 printf ("             illegalchar = 1; /* code is not in font */ \n");
00346 printf ("         } \n");
00347 printf ("     } \n");
00348 printf (" } \n");
00349 printf (" } \n");

```

```

00350     printf ("    if (illegalchar) totalwidth = -1;\n");
00351     printf ("\n");
00352     printf ("    return (totalwidth);\n");
00353     printf ("\n");
00354     printf ("}\n");
00355
00356     exit (EXIT_SUCCESS);
00357 }

```

5.24 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
00042 #include <stdio.h>
00043 #include <stdlib.h>
00044 #include <string.h>
00045
00046 #define MAXSTRING 256 ///< Maximum input line length - 1.
00047
00048 /* Definitions for Pikto in Plane 15 */
00049 #define PIKTO_START 0x0F0E70 ///< Start of Pikto code point range.
00050 #define PIKTO_END 0x0F11EF ///< End of Pikto code point range.
00051 /** Number of code points in Pikto range. */
00052 #define PIKTO_SIZE (PIKTO_END - PIKTO_START + 1)
00053
00054
00055 /**
00056  @brief The main function.
00057
00058  @param[in] argc The count of command line arguments.
00059  @param[in] argv Pointer to array of command line arguments.
00060  @return This program exits with status EXIT_SUCCESS.
00061  */
00062 int
00063 main (int argc, char **argv)

```

```

00064 {
00065
00066     int i; /* loop variable */
00067
00068     char teststring[MAXSTRING];
00069     int loc;
00070     char *gstart;
00071
00072     char glyph_width[0x20000];
00073     char pikto_width[PIKTO_SIZE];
00074
00075     FILE *infilefp;
00076
00077     if (argc != 3) {
00078         fprintf(stderr, "\n\nUsage: %s <unifont.hex> <combining.txt>\n\n", argv[0]);
00079         exit (EXIT_FAILURE);
00080     }
00081
00082     /*
00083      Read the collection of hex glyphs.
00084     */
00085     if ((infilefp = fopen (argv[1], "r")) == NULL) {
00086         fprintf (stderr, "ERROR - hex input file %s not found.\n\n", argv[1]);
00087         exit (EXIT_FAILURE);
00088     }
00089
00090     /* Flag glyph as non-existent until found. */
00091     memset (glyph_width, -1, 0x20000 * sizeof (char));
00092     memset (pikto_width, -1, (PIKTO_SIZE) * sizeof (char));
00093
00094     teststring[MAXSTRING-1] = '\0';
00095     while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
00096         sscanf (teststring, "%X:%s", &loc);
00097         if (loc < 0x20000) {
00098             gstart = strchr (teststring, ':') + 1;
00099             /*
00100              16 rows per glyph, 2 ASCII hexadecimal digits per byte,
00101              so divide number of digits by 32 (shift right 5 bits).
00102             */
00103             glyph_width[loc] = (strlen (gstart) - 1) » 5;
00104         }
00105         else if ((loc >= PIKTO_START) && (loc <= PIKTO_END)) {
00106             gstart = strchr (teststring, ':') + 1;
00107             pikto_width[loc - PIKTO_START] = strlen (gstart) <= 34 ? 1 : 2;
00108         }
00109     }
00110
00111     fclose (infilefp);
00112
00113     /*
00114      Now read the combining character code points. These have width of 0.
00115     */
00116     if ((infilefp = fopen (argv[2], "r")) == NULL) {
00117         fprintf (stderr, "ERROR - combining characters file %s not found.\n\n", argv[2]);
00118         exit (EXIT_FAILURE);
00119     }
00120
00121     while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
00122         sscanf (teststring, "%X:%s", &loc);
00123         if (loc < 0x20000) glyph_width[loc] = 0;
00124     }
00125
00126     fclose (infilefp);
00127
00128     /*
00129      Code Points with Unusual Properties (Unicode Standard, Chapter 4).
00130
00131      As of Unifont 10.0.04, use the widths in the "*-nonprinting.hex"
00132      files. If an application is smart enough to know how to handle
00133      these special cases, it will not render the "nonprinting" glyph
00134      and will treat the code point as being zero-width.
00135     */
00136     // glyph_width[0]=0; /* NULL character */
00137     // for (i = 0x0001; i <= 0x001F; i++) glyph_width[i]=-1; /* Control Characters */
00138     // for (i = 0x007F; i <= 0x009F; i++) glyph_width[i]=-1; /* Control Characters */
00139
00140     // glyph_width[0x034F]=0; /* combining grapheme joiner */
00141     // glyph_width[0x180B]=0; /* Mongolian free variation selector one */
00142     // glyph_width[0x180C]=0; /* Mongolian free variation selector two */
00143     // glyph_width[0x180D]=0; /* Mongolian free variation selector three */
00144     // glyph_width[0x180E]=0; /* Mongolian vowel separator */

```

```

00145 // glyph_width[0x200B]=0; /* zero width space */
00146 // glyph_width[0x200C]=0; /* zero width non-joiner */
00147 // glyph_width[0x200D]=0; /* zero width joiner */
00148 // glyph_width[0x200E]=0; /* left-to-right mark */
00149 // glyph_width[0x200F]=0; /* right-to-left mark */
00150 // glyph_width[0x202A]=0; /* left-to-right embedding */
00151 // glyph_width[0x202B]=0; /* right-to-left embedding */
00152 // glyph_width[0x202C]=0; /* pop directional formatting */
00153 // glyph_width[0x202D]=0; /* left-to-right override */
00154 // glyph_width[0x202E]=0; /* right-to-left override */
00155 // glyph_width[0x2060]=0; /* word joiner */
00156 // glyph_width[0x2061]=0; /* function application */
00157 // glyph_width[0x2062]=0; /* invisible times */
00158 // glyph_width[0x2063]=0; /* invisible separator */
00159 // glyph_width[0x2064]=0; /* invisible plus */
00160 // glyph_width[0x206A]=0; /* inhibit symmetric swapping */
00161 // glyph_width[0x206B]=0; /* activate symmetric swapping */
00162 // glyph_width[0x206C]=0; /* inhibit arabic form shaping */
00163 // glyph_width[0x206D]=0; /* activate arabic form shaping */
00164 // glyph_width[0x206E]=0; /* national digit shapes */
00165 // glyph_width[0x206F]=0; /* nominal digit shapes */
00166
00167 // /* Variation Selector-1 to Variation Selector-16 */
00168 // for (i = 0xFE00; i <= 0xFE0F; i++) glyph_width[i] = 0;
00169
00170 // glyph_width[0xFEFF]=0; /* zero width no-break space */
00171 // glyph_width[0xFF9]=0; /* interlinear annotation anchor */
00172 // glyph_width[0xFFA]=0; /* interlinear annotation separator */
00173 // glyph_width[0xFFB]=0; /* interlinear annotation terminator */
00174 /*
00175     Let glyph widths represent 0xFFFC (object replacement character)
00176     and 0xFFFD (replacement character).
00177 */
00178
00179 /*
00180     Hangul Jamo:
00181
00182     Leading Consonant (Choseong): leave spacing as is.
00183
00184     Hangul Choseong Filler (U+115F): set width to 2.
00185
00186     Hangul Jungseong Filler, Hangul Vowel (Jungseong), and
00187     Final Consonant (Jongseong): set width to 0, because these
00188     combine with the leading consonant as one composite syllabic
00189     glyph. As of Unicode 5.2, the Hangul Jamo block (U+1100..U+11FF)
00190     is completely filled.
00191 */
00192 // for (i = 0x1160; i <= 0x11FF; i++) glyph_width[i]=0; /* Vowels & Final Consonants */
00193
00194 /*
00195     Private Use Area -- the width is undefined, but likely
00196     to be 2 charcells wide either from a graphic glyph or
00197     from a four-digit hexadecimal glyph representing the
00198     code point. Therefore if any PUA glyph does not have
00199     a non-zero width yet, assign it a default width of 2.
00200     The Unicode Standard allows giving PUA characters
00201     default property values; see for example The Unicode
00202     Standard Version 5.0, p. 91. This same default is
00203     used for higher plane PUA code points below.
00204 */
00205 // for (i = 0xE000; i <= 0xF8FF; i++) {
00206 //     if (glyph_width[i] == 0) glyph_width[i]=2;
00207 // }
00208
00209 /*
00210     <not a character>
00211 */
00212 for (i = 0xFDD0; i <= 0xFDEF; i++) glyph_width[i] = -1;
00213 glyph_width[0xFFFE] = -1; /* Byte Order Mark */
00214 glyph_width[0xFFFF] = -1; /* Byte Order Mark */
00215
00216 /* Surrogate Code Points */
00217 for (i = 0xD800; i <= 0xDFFF; i++) glyph_width[i]=-1;
00218
00219 /* CJK Code Points */
00220 for (i = 0x4E00; i <= 0x9FFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00221 for (i = 0x3400; i <= 0x4DBF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00222 for (i = 0xF900; i <= 0xFAFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
00223
00224 /*
00225     Now generate the output file.

```

```

00226  */
00227  printf ("/*\n");
00228  printf (" wcswidth and wcswidth functions, as per IEEE 1003.1-2008\n");
00229  printf (" System Interfaces, pp. 2241 and 2251.\n\n");
00230  printf (" Author: Paul Hardy, 2013\n\n");
00231  printf (" Copyright (c) 2013 Paul Hardy\n\n");
00232  printf (" LICENSE:\n");
00233  printf ("\n");
00234  printf (" This program is free software: you can redistribute it and/or modify\n");
00235  printf (" it under the terms of the GNU General Public License as published by\n");
00236  printf (" the Free Software Foundation, either version 2 of the License, or\n");
00237  printf (" (at your option) any later version.\n");
00238  printf ("\n");
00239  printf (" This program is distributed in the hope that it will be useful,\n");
00240  printf (" but WITHOUT ANY WARRANTY; without even the implied warranty of\n");
00241  printf (" MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the\n");
00242  printf (" GNU General Public License for more details.\n");
00243  printf ("\n");
00244  printf (" You should have received a copy of the GNU General Public License\n");
00245  printf (" along with this program. If not, see <http://www.gnu.org/licenses/>.\n");
00246  printf ("*/\n\n");
00247
00248  printf ("#include <wchar.h>\n\n");
00249  printf ("/* Definitions for Pikto CSUR Private Use Area glyphs */\n");
00250  printf ("#define PIKTO_START\t0x%06X\n", PIKTO_START);
00251  printf ("#define PIKTO_END\t0x%06X\n", PIKTO_END);
00252  printf ("#define PIKTO_SIZE\t(PIKTO_END - PIKTO_START + 1)\n");
00253  printf ("\n\n");
00254  printf ("/* wcswidth -- return charcell positions of one code point */\n");
00255  printf ("inline int\ncwswidth (wchar_t wc)\n{\n");
00256  printf (" return (wcswidth (&wc, 1));\n");
00257  printf ("}\n");
00258  printf ("\n\n");
00259  printf ("int\ncwswidth (const wchar_t *pwcs, size_t n)\n{\n\n");
00260  printf (" int i; /* loop variable */\n");
00261  printf (" unsigned codept; /* Unicode code point of current character */\n");
00262  printf (" unsigned plane; /* Unicode plane, 0x00..0x10 */\n");
00263  printf (" unsigned lower17; /* lower 17 bits of Unicode code point */\n");
00264  printf (" unsigned lower16; /* lower 16 bits of Unicode code point */\n");
00265  printf (" int lowpt, midpt, highpt; /* for binary searching in plane1zeroes[] */\n");
00266  printf (" int found; /* for binary searching in plane1zeroes[] */\n");
00267  printf (" int totalwidth; /* total width of string, in charcells (1 or 2/glyph) */\n");
00268  printf (" int illegalchar; /* Whether or not this code point is illegal */\n");
00269  putchar ('\n');
00270
00271  /*
00272   Print the glyph_width[] array for glyphs widths in the
00273   Basic Multilingual Plane (Plane 0).
00274  */
00275  printf (" char glyph_width[0x20000] = {\n");
00276  for (i = 0; i < 0x10000; i++) {
00277   if ((i & 0x1F) == 0)
00278    printf ("\n /* U+%04X */ ", i);
00279   printf ("%d,", glyph_width[i]);
00280  }
00281  for (i = 0x10000; i < 0x20000; i++) {
00282   if ((i & 0x1F) == 0)
00283    printf ("\n /* U+%06X */ ", i);
00284   printf ("%d", glyph_width[i]);
00285   if (i < 0x1FFFF) putchar (',');
00286  }
00287  printf ("\n }; \n\n");
00288
00289  /*
00290   Print the pikto_width[] array for Pikto glyph widths.
00291  */
00292  printf (" char pikto_width[PIKTO_SIZE] = {\n");
00293  for (i = 0; i < PIKTO_SIZE; i++) {
00294   if ((i & 0x1F) == 0)
00295    printf ("\n /* U+%06X */ ", PIKTO_START + i);
00296   printf ("%d", pikto_width[i]);
00297   if ((PIKTO_START + i) < PIKTO_END) putchar (',');
00298  }
00299  printf ("\n }; \n\n");
00300
00301  /*
00302   Execution part of wcswidth.
00303  */
00304  printf ("\n");
00305  printf (" illegalchar = totalwidth = 0;\n");
00306  printf (" for (i = 0; !illegalchar && i < n; i++) {\n");

```

```

00307 printf (" codept = pwcs[i];\n");
00308 printf (" plane = codept > 16;\n");
00309 printf (" lower17 = codept & 0x1FFFF;\n");
00310 printf (" lower16 = codept & 0xFFFF;\n");
00311 printf (" if (plane < 2) { /* the most common case */\n");
00312 printf ("     if (glyph_width[lower17] < 0) illegalchar = 1;\n");
00313 printf ("     else totalwidth += glyph_width[lower17];\n");
00314 printf (" }\n");
00315 printf (" else { /* a higher plane or beyond Unicode range */\n");
00316 printf ("     if ((lower16 == 0xFFFE) || (lower16 == 0xFFFF)) {\n");
00317 printf ("         illegalchar = 1;\n");
00318 printf ("     }\n");
00319 printf ("     else if (plane < 4) { /* Ideographic Plane */\n");
00320 printf ("         totalwidth += 2; /* Default ideographic width */\n");
00321 printf ("     }\n");
00322 printf ("     else if (plane == 0x0F) { /* CSUR Private Use Area */\n");
00323 printf ("         if (lower16 <= 0x0E6F) { /* Kinya */\n");
00324 printf ("             totalwidth++; /* all Kinya syllables have width 1 */\n");
00325 printf ("         }\n");
00326 printf ("         else if (lower16 <= (PIKTO_END & 0xFFFF)) { /* Pikto */\n");
00327 printf ("             if (pikto_width[lower16 - (PIKTO_START & 0xFFFF)] < 0) illegalchar = 1;\n");
00328 printf ("             else totalwidth += pikto_width[lower16 - (PIKTO_START & 0xFFFF)];\n");
00329 printf ("         }\n");
00330 printf ("     }\n");
00331 printf ("     else if (plane > 0x10) {\n");
00332 printf ("         illegalchar = 1;\n");
00333 printf ("     }\n");
00334 printf ("     /* Other non-printing in higher planes; return -1 as per IEEE 1003.1-2008. */\n");
00335 printf ("     else if (/* language tags */\n");
00336 printf ("         codept == 0x0E0001 || (codept >= 0x0E0020 && codept <= 0x0E007F) ||\n");
00337 printf ("         /* variation selectors, 0x0E0100..0x0E01EF */\n");
00338 printf ("         (codept >= 0x0E0100 && codept <= 0x0E01EF)) {\n");
00339 printf ("             illegalchar = 1;\n");
00340 printf ("         }\n");
00341 printf ("     /*\n");
00342 printf ("         Unicode plane 0x02..0x10 printing character\n");
00343 printf ("     */\n");
00344 printf ("     else {\n");
00345 printf ("         illegalchar = 1; /* code is not in font */\n");
00346 printf ("     }\n");
00347 printf (" }\n");
00348 printf (" }\n");
00349 printf (" }\n");
00350 printf (" if (illegalchar) totalwidth = -1;\n");
00351 printf (" }\n");
00352 printf (" return (totalwidth);\n");
00353 printf (" }\n");
00354 printf (" }\n");
00355
00356 exit (EXIT_SUCCESS);
00357 }

```

5.25 src/unihex2bmp.c File Reference

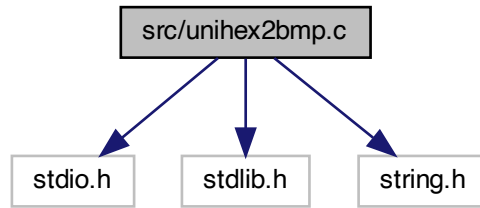
unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points into a bitmap for editing

```

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

```

Include dependency graph for unihex2bmp.c:



Macros

- `#define` [MAXBUF](#) 256

Functions

- `int` [main](#) (`int` argc, `char` *argv[])
The main function.
- `int` [hex2bit](#) (`char` *instring, `unsigned char` character[32][4])
Generate a bitmap for one glyph.
- `int` [init](#) (`unsigned char` bitmap[17 * 32][18 * 4])
Initialize the bitmap grid.

Variables

- `char` * [hex](#) [18]
GNU Unifont bitmaps for hexadecimal digits.
- `unsigned char` [hexbits](#) [18][32]
The digits converted into bitmaps.
- `unsigned` [unipage](#) = 0
Unicode page number, 0x00..0xff.
- `int` [flip](#) = 1
Transpose entire matrix as in Unicode book.

5.25.1 Detailed Description

unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points into a bitmap for editing

Author

Paul Hardy, [unifoundry <at> unifoundry.com](mailto:unifoundry@unifoundry.com), December 2007

Copyright

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

This program reads in a GNU Unifont .hex file, extracts a range of 256 code points, and converts it a Microsoft Bitmap (.bmp) or Wireless Bitmap file.

Synopsis: unihex2bmp [-iin_file.hex] [-oout_file.bmp] [-f] [-phex_page_num] [-w]

Definition in file [unihex2bmp.c](#).

5.25.2 Macro Definition Documentation

5.25.2.1 MAXBUF

```
#define MAXBUF 256
```

Definition at line [47](#) of file [unihex2bmp.c](#).

5.25.3 Function Documentation

5.25.3.1 hex2bit()

```
int hex2bit (
    char * instring,
    unsigned char character[32][4] )
```

Generate a bitmap for one glyph.

Convert the portion of a hex string after the ':' into a character bitmap.

If string is ≥ 128 characters, it will fill all 4 bytes per row. If string is ≥ 64 characters and < 128 , it will fill 2 bytes per row. Otherwise, it will fill 1 byte per row.

Parameters

in	instring	The character array containing the glyph bitmap.
out	character	Glyph bitmap, 8, 16, or 32 columns by 16 rows tall.

Returns

Always returns 0.

Definition at line [361](#) of file [unihex2bmp.c](#).

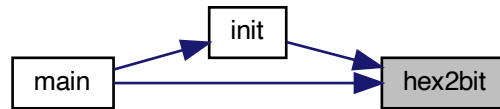
```

00362 {
00363
00364     int i; /* current row in bitmap character */
00365     int j; /* current character in input string */
00366     int k; /* current byte in bitmap character */
00367     int width; /* number of output bytes to fill - 1: 0, 1, 2, or 3 */
00368
00369     for (i=0; i<32; i++) /* erase previous character */
00370         character[i][0] = character[i][1] = character[i][2] = character[i][3] = 0;
00371     j=0; /* current location is at beginning of instring */
00372
00373     if (strlen (instring) <= 34) /* 32 + possible '\r', '\n' */
00374         width = 0;
00375     else if (strlen (instring) <= 66) /* 64 + possible '\r', '\n' */
00376         width = 1;
00377     else if (strlen (instring) <= 98) /* 96 + possible '\r', '\n' */
00378         width = 3;
00379     else /* the maximum allowed is quadruple-width */
00380         width = 4;
00381
00382     k = (width > 1) ? 0 : 1; /* if width > double, start at index 1 else at 0 */
00383
00384     for (i=8; i<24; i++) { /* 16 rows per input character, rows 8..23 */
00385         sscanf (&instring[j], "%2hhx", &character[i][k]);
00386         j += 2;
00387         if (width > 0) { /* add next pair of hex digits to this row */
00388             sscanf (&instring[j], "%2hhx", &character[i][k+1]);
00389             j += 2;
00390             if (width > 1) { /* add next pair of hex digits to this row */
00391                 sscanf (&instring[j], "%2hhx", &character[i][k+2]);
00392                 j += 2;
00393                 if (width > 2) { /* quadruple-width is maximum width */
00394                     sscanf (&instring[j], "%2hhx", &character[i][k+3]);
00395                     j += 2;
00396                 }
00397             }
00398         }
00399     }
00400 }
00401     return (0);

```

```
00402 }
```

Here is the caller graph for this function:



5.25.3.2 init()

```
int init (
    unsigned char bitmap[17 * 32][18 * 4] )
```

Initialize the bitmap grid.

Parameters

out	bitmap	The bitmap to gen- erate, with 32x32 pixel glyph areas.

Returns

Always returns 0.

Definition at line [412](#) of file [unihex2bmp.c](#).

```

00413 {
00414     int i, j;
00415     unsigned char charbits[32][4]; /* bitmap for one character, 4 bytes/row */
00416     unsigned toppixelrow;
00417     unsigned thiscol;
00418     unsigned char pnybble0, pnybble1, pnybble2, pnybble3;
00419
00420     for (i=0; i<18; i++) { /* bitmaps for '0'..'9', 'A'..'F', 'u', '+' */
00421
00422         hex2bit (&hex[i][5], charbits); /* convert hex string to 32*4 bitmap */
```

```

00423
00424     for (j=0; j<32; j++) hexbits[i][j] = ~charbits[j][1];
00425 }
00426
00427 /*
00428  Initialize bitmap to all white.
00429 */
00430 for (toppixelrow=0; toppixelrow < 17*32; toppixelrow++) {
00431     for (thiscol=0; thiscol<18; thiscol++) {
00432         bitmap[toppixelrow][thiscol < 2] = 0xff;
00433         bitmap[toppixelrow][thiscol < 2 | 1] = 0xff;
00434         bitmap[toppixelrow][thiscol < 2 | 2] = 0xff;
00435         bitmap[toppixelrow][thiscol < 2 | 3] = 0xff;
00436     }
00437 }
00438 /*
00439  Write the "u+nnnn" table header in the upper left-hand corner,
00440  where nnnn is the upper 16 bits of a 32-bit Unicode assignment.
00441 */
00442 pnybble3 = (unipage >> 20);
00443 pnybble2 = (unipage >> 16) & 0xf;
00444 pnybble1 = (unipage >> 12) & 0xf;
00445 pnybble0 = (unipage >> 8) & 0xf;
00446 for (i=0; i<32; i++) {
00447     bitmap[i][1] = hexbits[16][i]; /* copy 'u' */
00448     bitmap[i][2] = hexbits[17][i]; /* copy '+' */
00449     bitmap[i][3] = hexbits[pnybble3][i];
00450     bitmap[i][4] = hexbits[pnybble2][i];
00451     bitmap[i][5] = hexbits[pnybble1][i];
00452     bitmap[i][6] = hexbits[pnybble0][i];
00453 }
00454 /*
00455  Write low-order 2 bytes of Unicode number assignments, as hex labels
00456 */
00457 pnybble3 = (unipage >> 4) & 0xf; /* Highest-order hex digit */
00458 pnybble2 = (unipage >> 0) & 0xf; /* Next highest-order hex digit */
00459 /*
00460  Write the column headers in bitmap[] (row headers if flipped)
00461 */
00462 toppixelrow = 32 * 17 - 1; /* maximum pixel row number */
00463 /*
00464  Label the column headers. The hexbits[] bytes are split across two
00465  bitmap[] entries to center a the hex digits in a column of 4 bytes.
00466  OR highest byte with 0xf0 and lowest byte with 0x0f to make outer
00467  nybbles white (0=black, 1=white).
00468 */
00469 for (i=0; i<16; i++) {
00470     for (j=0; j<32; j++) {
00471         if (flip) { /* transpose matrix */
00472             bitmap[j][((i+2) < 2) | 0] = (hexbits[pnybble3][j] >> 4) | 0xf0;
00473             bitmap[j][((i+2) < 2) | 1] = (hexbits[pnybble3][j] << 4) |
00474                 (hexbits[pnybble2][j] >> 4);
00475             bitmap[j][((i+2) < 2) | 2] = (hexbits[pnybble2][j] << 4) |
00476                 (hexbits[i][j] >> 4);
00477             bitmap[j][((i+2) < 2) | 3] = (hexbits[i][j] << 4) | 0x0f;
00478         }
00479         else {
00480             bitmap[j][((i+2) < 2) | 1] = (hexbits[i][j] >> 4) | 0xf0;
00481             bitmap[j][((i+2) < 2) | 2] = (hexbits[i][j] << 4) | 0x0f;
00482         }
00483     }
00484 }
00485 /*
00486  Now use the single hex digit column graphics to label the row headers.
00487 */
00488 for (i=0; i<16; i++) {
00489     toppixelrow = 32 * (i + 1) - 1; /* from bottom to top */
00490
00491     for (j=0; j<32; j++) {
00492         if (!flip) { /* if not transposing matrix */
00493             bitmap[toppixelrow + j][4] = hexbits[pnybble3][j];
00494             bitmap[toppixelrow + j][5] = hexbits[pnybble2][j];
00495         }
00496         bitmap[toppixelrow + j][6] = hexbits[i][j];
00497     }
00498 }
00499 /*
00500  Now draw grid lines in bitmap, around characters we just copied.
00501 */
00502 /* draw vertical lines 2 pixels wide */
00503 for (i=1*32; i<17*32; i++) {

```

```

00504     if ((i & 0x1f) == 7)
00505         i++;
00506     else if ((i & 0x1f) == 14)
00507         i += 2;
00508     else if ((i & 0x1f) == 22)
00509         i++;
00510     for (j=1; j<18; j++) {
00511         bitmap[i][(j << 2) | 3] &= 0xfe;
00512     }
00513 }
00514 /* draw horizontal lines 1 pixel tall */
00515 for (i=1*32-1; i<18*32-1; i+=32) {
00516     for (j=2; j<18; j++) {
00517         bitmap[i][(j << 2) ] = 0x00;
00518         bitmap[i][(j << 2) | 1] = 0x81;
00519         bitmap[i][(j << 2) | 2] = 0x81;
00520         bitmap[i][(j << 2) | 3] = 0x00;
00521     }
00522 }
00523 /* fill in top left corner pixel of grid */
00524 bitmap[31][7] = 0xfe;
00525
00526 return (0);
00527 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



5.25.3.3 main()

```

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

```

The main function.

Parameters

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

Returns

This program exits with status 0.

Definition at line 96 of file [unihex2bmp.c](#).

```

00097 {
00098
00099     int i, j;                /* loop variables */
00100     unsigned k0;             /* temp Unicode char variable */
00101     unsigned swap;           /* temp variable for swapping values */
00102     char inbuf[256];         /* input buffer */
00103     unsigned filesize;       /* size of file in bytes */
00104     unsigned bitmapsizes;    /* size of bitmap image in bytes */
00105     unsigned thischar;       /* the current character */
00106     unsigned char thischarbyte; /* unsigned char lowest byte of Unicode char */
00107     int thischarrow;         /* row 0..15 where this character belongs */
00108     int thiscol;             /* column 0..15 where this character belongs */
00109     int toppixelrow;         /* pixel row, 0..16*32-1 */
00110     unsigned lastpage=0;     /* the last Unicode page read in font file */
00111     int wbmp=0;              /* set to 1 if writing .wbmp format file */
00112
00113     unsigned char bitmap[17*32][18*4]; /* final bitmap */
00114     unsigned char charbits[32][4]; /* bitmap for one character, 4 bytes/row */
00115
00116     char *infile="", *outfile=""; /* names of input and output files */
00117     FILE *infp, *outfp; /* file pointers of input and output files */
00118
00119     int init();              /* initializes bitmap row/col labeling, &c. */
00120     int hex2bit();           /* convert hex string --> bitmap */
00121
00122     bitmapsizes = 17*32*18*4; /* 17 rows by 18 cols, each 4 bytes */
00123
00124     if (argc > 1) {
00125         for (i = 1; i < argc; i++) {
00126             if (argv[i][0] == '-') { /* this is an option argument */
00127                 switch (argv[i][1]) {
00128                     case 'f': /* flip (transpose) glyphs in bitmap as in standard */
00129                         flip = !flip;
00130                         break;
00131                     case 'i': /* name of input file */
00132                         infile = &argv[i][2];
00133                         break;
00134                     case 'o': /* name of output file */
00135                         outfile = &argv[i][2];
00136                         break;
00137                     case 'p': /* specify a Unicode page other than default of 0 */
00138                         sscanf (&argv[i][2], "%x", &unipage); /* Get Unicode page */
00139                         break;

```

```

00140         case 'w': /* write a .wbmp file instead of a .bmp file */
00141             wbmp = 1;
00142             break;
00143         default: /* if unrecognized option, print list and exit */
00144             fprintf(stderr, "\nSyntax:\n\n");
00145             fprintf(stderr, " %s -p<Unicode_Page> ", argv[0]);
00146             fprintf(stderr, "-i<Input_File> -o<Output_File> -w\n\n");
00147             fprintf(stderr, " -w specifies .wbmp output instead of ");
00148             fprintf(stderr, "default Windows .bmp output.\n\n");
00149             fprintf(stderr, " -p is followed by 1 to 6 ");
00150             fprintf(stderr, "Unicode page hex digits ");
00151             fprintf(stderr, "(default is Page 0).\n\n");
00152             fprintf(stderr, "\nExample:\n\n");
00153             fprintf(stderr, " %s -p83 -iunifont.hex -ou83.bmp\n\n\n",
00154                 argv[0]);
00155             exit (1);
00156     }
00157 }
00158 }
00159 }
00160 /*
00161  Make sure we can open any I/O files that were specified before
00162  doing anything else.
00163 */
00164 if (strlen (infile) > 0) {
00165     if ((infp = fopen (infile, "r")) == NULL) {
00166         fprintf (stderr, "Error: can't open %s for input.\n", infile);
00167         exit (1);
00168     }
00169 }
00170 else {
00171     infp = stdin;
00172 }
00173 if (strlen (outfile) > 0) {
00174     if ((outfp = fopen (outfile, "w")) == NULL) {
00175         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00176         exit (1);
00177     }
00178 }
00179 else {
00180     outfp = stdout;
00181 }
00182 (void)init(bitmap); /* initialize bitmap with row/column headers, etc. */
00183
00184 /*
00185  Read in the characters in the page
00186 */
00187 while (lastpage <= unipage && fgetc (inbuf, MAXBUF-1, infp) != NULL) {
00188     sscanf (inbuf, "%x", &thischar);
00189     lastpage = thischar » 8; /* keep Unicode page to see if we can stop */
00190     if (lastpage == unipage) {
00191         thischarbyte = (unsigned char)(thischar & 0xff);
00192         for (k0=0; inbuf[k0] != ':'; k0++);
00193         k0++;
00194         hex2bit (&inbuf[k0], charbits); /* convert hex string to 32*4 bitmap */
00195
00196         /*
00197          Now write character bitmap upside-down in page array, to match
00198          .bmp file order. In the .wbmp' and .bmp files, white is a '1'
00199          bit and black is a '0' bit, so complement charbits[].
00200          */
00201
00202         thiscol = (thischarbyte & 0xf) + 2; /* column number will be 1..16 */
00203         thischarrow = thischarbyte » 4; /* character row number, 0..15 */
00204         if (flip) { /* swap row and column placement */
00205             swap = thiscol;
00206             thiscol = thischarrow;
00207             thischarrow = swap;
00208             thiscol += 2; /* column index starts at 1 */
00209             thischarrow -= 2; /* row index starts at 0 */
00210         }
00211         toppixelrow = 32 * (thischarrow + 1) - 1; /* from bottom to top */
00212
00213         /*
00214          Copy the center of charbits[] because hex characters only
00215          occupy rows 8 to 23 and column byte 2 (and for 16 bit wide
00216          characters, byte 3). The charbits[] array was given 32 rows
00217          and 4 column bytes for completeness in the beginning.
00218          */
00219         for (i=8; i<24; i++) {
00220

```

```

00221     bitmap[toppixelrow + i][ (thiscol « 2) | 0] =
00222     ~charbits[i][0] & 0xff;
00223     bitmap[toppixelrow + i][ (thiscol « 2) | 1] =
00224     ~charbits[i][1] & 0xff;
00225     bitmap[toppixelrow + i][ (thiscol « 2) | 2] =
00226     ~charbits[i][2] & 0xff;
00227     /* Only use first 31 bits; leave vertical rule in 32nd column */
00228     bitmap[toppixelrow + i][ (thiscol « 2) | 3] =
00229     ~charbits[i][3] & 0xfe;
00230 }
00231 /*
00232  Leave white space in 32nd column of rows 8, 14, 15, and 23
00233  to leave 16 pixel height upper, middle, and lower guides.
00234 */
00235 bitmap[toppixelrow + 8][ (thiscol « 2) | 3] |= 1;
00236 bitmap[toppixelrow + 14][ (thiscol « 2) | 3] |= 1;
00237 bitmap[toppixelrow + 15][ (thiscol « 2) | 3] |= 1;
00238 bitmap[toppixelrow + 23][ (thiscol « 2) | 3] |= 1;
00239 }
00240 }
00241 /*
00242  Now write the appropriate bitmap file format, either
00243  Wireless Bitmap or Microsoft Windows bitmap.
00244 */
00245 if (wbmp) { /* Write a Wireless Bitmap .wbmp format file */
00246     /*
00247      Write WBMP header
00248     */
00249     fprintf (outfp, "%c", 0x00); /* Type of image; always 0 (monochrome) */
00250     fprintf (outfp, "%c", 0x00); /* Reserved; always 0 */
00251     fprintf (outfp, "%c%c", 0x84, 0x40); /* Width = 576 pixels */
00252     fprintf (outfp, "%c%c", 0x84, 0x20); /* Height = 544 pixels */
00253     /*
00254      Write bitmap image
00255     */
00256     for (toppixelrow=0; toppixelrow <= 17*32-1; toppixelrow++) {
00257         for (j=0; j<18; j++) {
00258             fprintf (outfp, "%c", bitmap[toppixelrow][ (j«2)   ]);
00259             fprintf (outfp, "%c", bitmap[toppixelrow][ (j«2) | 1]);
00260             fprintf (outfp, "%c", bitmap[toppixelrow][ (j«2) | 2]);
00261             fprintf (outfp, "%c", bitmap[toppixelrow][ (j«2) | 3]);
00262         }
00263     }
00264 }
00265 else { /* otherwise, write a Microsoft Windows .bmp format file */
00266     /*
00267      Write the .bmp file -- start with the header, then write the bitmap
00268     */
00269     /* 'B', 'M' appears at start of every .bmp file */
00270     fprintf (outfp, "%c%c", 0x42, 0x4d);
00271
00272     /* Write file size in bytes */
00273     filesize = 0x3E + bitmapsize;
00274     fprintf (outfp, "%c", (unsigned char)((filesize >> 0x0f) & 0xff));
00275     fprintf (outfp, "%c", (unsigned char)((filesize >> 0x08) & 0xff));
00276     fprintf (outfp, "%c", (unsigned char)((filesize >> 0x01) & 0xff));
00277     fprintf (outfp, "%c", (unsigned char)((filesize >> 0x10) & 0xff));
00278     fprintf (outfp, "%c", (unsigned char)((filesize >> 0x18) & 0xff));
00279
00280     /* Reserved - 0's */
00281     fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00282
00283     /* Offset from start of file to bitmap data */
00284     fprintf (outfp, "%c%c%c%c", 0x3E, 0x00, 0x00, 0x00);
00285
00286     /* Length of bitmap info header */
00287     fprintf (outfp, "%c%c%c%c", 0x28, 0x00, 0x00, 0x00);
00288
00289     /* Width of bitmap in pixels */
00290     fprintf (outfp, "%c%c%c%c", 0x40, 0x02, 0x00, 0x00);
00291
00292     /* Height of bitmap in pixels */
00293     fprintf (outfp, "%c%c%c%c", 0x20, 0x02, 0x00, 0x00);
00294
00295     /* Planes in bitmap (fixed at 1) */
00296     fprintf (outfp, "%c%c", 0x01, 0x00);
00297
00298     /* bits per pixel (1 = monochrome) */
00299     fprintf (outfp, "%c%c", 0x01, 0x00);
00300
00301     /* Compression (0 = none) */

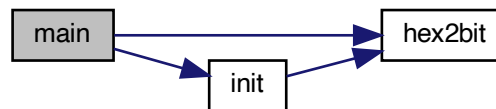
```

```

00302     fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00303
00304     /* Size of bitmap data in bytes */
00305     fprintf (outfp, "%c", (unsigned char)((bitmapsizesize & 0xff));
00306     fprintf (outfp, "%c", (unsigned char)((bitmapsizesize >> 0x08) & 0xff));
00307     fprintf (outfp, "%c", (unsigned char)((bitmapsizesize >> 0x10) & 0xff));
00308     fprintf (outfp, "%c", (unsigned char)((bitmapsizesize >> 0x18) & 0xff));
00309
00310     /* Horizontal resolution in pixels per meter */
00311     fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);
00312
00313     /* Vertical resolution in pixels per meter */
00314     fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);
00315
00316     /* Number of colors used */
00317     fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);
00318
00319     /* Number of important colors */
00320     fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);
00321
00322     /* The color black: B=0x00, G=0x00, R=0x00, Filler=0xFF */
00323     fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00324
00325     /* The color white: B=0xFF, G=0xFF, R=0xFF, Filler=0xFF */
00326     fprintf (outfp, "%c%c%c%c", 0xFF, 0xFF, 0xFF, 0x00);
00327
00328     /*
00329     Now write the raw data bits.  Data is written from the lower
00330     left-hand corner of the image to the upper right-hand corner
00331     of the image.
00332     */
00333     for (toppixelrow=17*32-1; topixelrow >= 0; topixelrow--) {
00334         for (j=0; j<18; j++) {
00335             fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2)   ]);
00336             fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 1]);
00337             fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 2]);
00338
00339             fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 3]);
00340         }
00341     }
00342 }
00343 exit (0);
00344 }

```

Here is the call graph for this function:



5.25.4 Variable Documentation

5.25.4.1 flip

```
int flip =1
```


Transpose entire matrix as in Unicode book.

Definition at line 85 of file [unihex2bmp.c](#).

5.25.4.2 hex

char* hex[18]

Initial value:

```
= {
    "0030:000000001824424242424224180000",
    "0031:0000000008182808080808083E0000",
    "0032:0000000003C4242020C102040407E0000",
    "0033:0000000003C4242021C020242423C0000",
    "0034:00000000040C142444447E0404040000",
    "0035:0000000007E4040407C020202423C0000",
    "0036:000000001C2040407C424242423C0000",
    "0037:0000000007E020204040808080000",
    "0038:0000000003C4242423C424242423C0000",
    "0039:0000000003C4242423E02020204380000",
    "0041:0000000018242442427E424242420000",
    "0042:0000000007C4242427C424242427C0000",
    "0043:0000000003C42424040404042423C0000",
    "0044:0000000078444242424242424780000",
    "0045:0000000007E4040407C404040407E0000",
    "0046:0000000007E4040407C40404040400000",
    "0055:0000000042424242424242423C0000",
    "002B:0000000000000808087F080808000000"
}
```

GNU Unifont bitmaps for hexadecimal digits.

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

Looking at the final bitmap as a grid of 32*32 bit tiles, the first row contains a hexadecimal character string of the first 3 hex digits in a 4 digit Unicode character name; the top column contains a hex character string of the 4th (low-order) hex digit of the Unicode character.

Definition at line 62 of file [unihex2bmp.c](#).

5.25.4.3 hexbits

unsigned char hexbits[18][32]

The digits converted into bitmaps.

Definition at line 82 of file [unihex2bmp.c](#).

5.25.4.4 unipage

unsigned unipage =0

Unicode page number, 0x00..0xff.

Definition at line 84 of file [unihex2bmp.c](#).

5.26 unihex2bmp.c

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

```
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
00043  #include <stdio.h>
00044  #include <stdlib.h>
00045  #include <string.h>
00046
00047  #define MAXBUF 256
00048
00049
00050  /**
00051  @brief GNU Unifont bitmaps for hexadecimal digits.
00052
00053  These are the GNU Unifont hex strings for '0'-'9' and 'A'-'F',
00054  for encoding as bit strings in row and column headers.
00055
00056  Looking at the final bitmap as a grid of 32*32 bit tiles, the
00057  first row contains a hexadecimal character string of the first
00058  3 hex digits in a 4 digit Unicode character name; the top column
00059  contains a hex character string of the 4th (low-order) hex digit
00060  of the Unicode character.
00061  */
```

```

00062 char *hex[18]= {
00063     "0030:0000000018244242424242424180000", /* Hex digit 0 */
00064     "0031:000000000818280808080808083E0000", /* Hex digit 1 */
00065     "0032:000000003C4242020C102040407E0000", /* Hex digit 2 */
00066     "0033:000000003C4242021C020242423C0000", /* Hex digit 3 */
00067     "0034:00000000040C142444447E0404040000", /* Hex digit 4 */
00068     "0035:000000007E4040407C020202423C0000", /* Hex digit 5 */
00069     "0036:000000001C2040407C424242423C0000", /* Hex digit 6 */
00070     "0037:000000007E0202040404080808080000", /* Hex digit 7 */
00071     "0038:000000003C4242423C424242423C0000", /* Hex digit 8 */
00072     "0039:000000003C4242423E02020204380000", /* Hex digit 9 */
00073     "0041:0000000018242442427E424242420000", /* Hex digit A */
00074     "0042:000000007C4242427C424242427C0000", /* Hex digit B */
00075     "0043:000000003C42424040404042423C0000", /* Hex digit C */
00076     "0044:0000000078444242424242424780000", /* Hex digit D */
00077     "0045:000000007E4040407C404040407E0000", /* Hex digit E */
00078     "0046:000000007E4040407C404040400000", /* Hex digit F */
00079     "0055:0000000042424242424242423C0000", /* Unicode 'U' */
00080     "002B:0000000000000808087F080808000000", /* Unicode '+' */
00081 };
00082 unsigned char hexbits[18][32]; ///< The digits converted into bitmaps.
00083
00084 unsigned unipage=0; ///< Unicode page number, 0x00..0xff.
00085 int flip=1;          ///< Transpose entire matrix as in Unicode book.
00086
00087 /**
00088  * @brief The main function.
00089  *
00090  * @param[in] argc The count of command line arguments.
00091  * @param[in] argv Pointer to array of command line arguments.
00092  * @return This program exits with status 0.
00093  */
00094
00095 int
00096 main (int argc, char *argv[])
00097 {
00098     int i, j; /* loop variables */
00099     unsigned k0; /* temp Unicode char variable */
00100     unsigned swap; /* temp variable for swapping values */
00101     char inbuf[256]; /* input buffer */
00102     unsigned filesize; /* size of file in bytes */
00103     unsigned bitmapsize; /* size of bitmap image in bytes */
00104     unsigned thischar; /* the current character */
00105     unsigned char thischarbyte; /* unsigned char lowest byte of Unicode char */
00106     int thischarrow; /* row 0..15 where this character belongs */
00107     int thiscol; /* column 0..15 where this character belongs */
00108     int toppixelrow; /* pixel row, 0..16*32-1 */
00109     unsigned lastpage=0; /* the last Unicode page read in font file */
00110     int wbmp=0; /* set to 1 if writing .wbmp format file */
00111
00112     unsigned char bitmap[17*32][18*4]; /* final bitmap */
00113     unsigned char charbits[32][4]; /* bitmap for one character, 4 bytes/row */
00114
00115     char *infile="", *outfile=""; /* names of input and output files */
00116     FILE *infp, *outfp; /* file pointers of input and output files */
00117
00118     int init(); /* initializes bitmap row/col labeling, &c. */
00119     int hex2bit(); /* convert hex string --> bitmap */
00120
00121     bitmapsize = 17*32*18*4; /* 17 rows by 18 cols, each 4 bytes */
00122
00123     if (argc > 1) {
00124         for (i = 1; i < argc; i++) {
00125             if (argv[i][0] == '-') { /* this is an option argument */
00126                 switch (argv[i][1]) {
00127                     case 'f': /* flip (transpose) glyphs in bitmap as in standard */
00128                         flip = !flip;
00129                         break;
00130                     case 'i': /* name of input file */
00131                         infile = &argv[i][2];
00132                         break;
00133                     case 'o': /* name of output file */
00134                         outfile = &argv[i][2];
00135                         break;
00136                     case 'p': /* specify a Unicode page other than default of 0 */
00137                         sscanf (&argv[i][2], "%x", &unipage); /* Get Unicode page */
00138                         break;
00139                     case 'w': /* write a .wbmp file instead of a .bmp file */
00140                         wbmp = 1;
00141                         break;

```

```

00143         default: /* if unrecognized option, print list and exit */
00144             fprintf (stderr, "\nSyntax:\n\n");
00145             fprintf (stderr, "  %s -p<Unicode_Page> ", argv[0]);
00146             fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
00147             fprintf (stderr, "  -w specifies .wbmp output instead of ");
00148             fprintf (stderr, "default Windows .bmp output.\n\n");
00149             fprintf (stderr, "  -p is followed by 1 to 6 ");
00150             fprintf (stderr, "Unicode page hex digits ");
00151             fprintf (stderr, "(default is Page 0).\n\n");
00152             fprintf (stderr, "\nExample:\n\n");
00153             fprintf (stderr, "  %s -p83 -iunifont.hex -ou83.bmp\n\n",
00154                     argv[0]);
00155             exit (1);
00156         }
00157     }
00158 }
00159 }
00160 /*
00161  Make sure we can open any I/O files that were specified before
00162  doing anything else.
00163  */
00164 if (strlen (infile) > 0) {
00165     if ((infp = fopen (infile, "r")) == NULL) {
00166         fprintf (stderr, "Error: can't open %s for input.\n", infile);
00167         exit (1);
00168     }
00169 }
00170 else {
00171     infp = stdin;
00172 }
00173 if (strlen (outfile) > 0) {
00174     if ((outfp = fopen (outfile, "w")) == NULL) {
00175         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
00176         exit (1);
00177     }
00178 }
00179 else {
00180     outfp = stdout;
00181 }
00182 (void)init(bitmap); /* initialize bitmap with row/column headers, etc. */
00183
00184 /*
00185  Read in the characters in the page
00186  */
00187 while (lastpage <= unipage && fgetc (inbuf, MAXBUF-1, infp) != NULL) {
00188     sscanf (inbuf, "%x", &thischar);
00189     lastpage = thischar » 8; /* keep Unicode page to see if we can stop */
00190     if (lastpage == unipage) {
00191         thischarbyte = (unsigned char)(thischar & 0xff);
00192         for (k0=0; inbuf[k0] != ':'; k0++);
00193         k0++;
00194         hex2bit (&inbuf[k0], charbits); /* convert hex string to 32*4 bitmap */
00195
00196         /*
00197          Now write character bitmap upside-down in page array, to match
00198          .bmp file order. In the .wbmp' and .bmp files, white is a '1'
00199          bit and black is a '0' bit, so complement charbits[].
00200          */
00201
00202         thiscol = (thischarbyte & 0xf) + 2; /* column number will be 1..16 */
00203         thischarrow = thischarbyte » 4; /* character row number, 0..15 */
00204         if (flip) { /* swap row and column placement */
00205             swap = thiscol;
00206             thiscol = thischarrow;
00207             thischarrow = swap;
00208             thiscol += 2; /* column index starts at 1 */
00209             thischarrow -= 2; /* row index starts at 0 */
00210         }
00211         toppixelrow = 32 * (thischarrow + 1) - 1; /* from bottom to top */
00212
00213         /*
00214          Copy the center of charbits[] because hex characters only
00215          occupy rows 8 to 23 and column byte 2 (and for 16 bit wide
00216          characters, byte 3). The charbits[] array was given 32 rows
00217          and 4 column bytes for completeness in the beginning.
00218          */
00219         for (i=8; i<24; i++) {
00220             bitmap[toppixelrow + i][(thiscol « 2) | 0] =
00221                 ~charbits[i][0] & 0xff;
00222             bitmap[toppixelrow + i][(thiscol « 2) | 1] =

```

```

00224         ~charbits[i][1] & 0xff;
00225         bitmap[toppixelrow + i][(thiscol « 2) | 2] =
00226         ~charbits[i][2] & 0xff;
00227         /* Only use first 31 bits; leave vertical rule in 32nd column */
00228         bitmap[toppixelrow + i][(thiscol « 2) | 3] =
00229         ~charbits[i][3] & 0xfe;
00230     }
00231     /*
00232     Leave white space in 32nd column of rows 8, 14, 15, and 23
00233     to leave 16 pixel height upper, middle, and lower guides.
00234     */
00235     bitmap[toppixelrow + 8][(thiscol « 2) | 3] |= 1;
00236     bitmap[toppixelrow + 14][(thiscol « 2) | 3] |= 1;
00237     bitmap[toppixelrow + 15][(thiscol « 2) | 3] |= 1;
00238     bitmap[toppixelrow + 23][(thiscol « 2) | 3] |= 1;
00239 }
00240 }
00241 /*
00242 Now write the appropriate bitmap file format, either
00243 Wireless Bitmap or Microsoft Windows bitmap.
00244 */
00245 if (wbmp) { /* Write a Wireless Bitmap .wbmp format file */
00246     /*
00247     Write WBMP header
00248     */
00249     fprintf (outfp, "%c", 0x00); /* Type of image; always 0 (monochrome) */
00250     fprintf (outfp, "%c", 0x00); /* Reserved; always 0 */
00251     fprintf (outfp, "%c%c", 0x84, 0x40); /* Width = 576 pixels */
00252     fprintf (outfp, "%c%c", 0x84, 0x20); /* Height = 544 pixels */
00253     /*
00254     Write bitmap image
00255     */
00256     for (toppixelrow=0; toppixelrow <= 17*32-1; toppixelrow++) {
00257         for (j=0; j<18; j++) {
00258             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 0]);
00259             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 1]);
00260             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 2]);
00261             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 3]);
00262         }
00263     }
00264 }
00265 else { /* otherwise, write a Microsoft Windows .bmp format file */
00266     /*
00267     Write the .bmp file -- start with the header, then write the bitmap
00268     */
00269     /* 'B', 'M' appears at start of every .bmp file */
00270     fprintf (outfp, "%c%c", 0x42, 0x4d);
00271
00272     /* Write file size in bytes */
00273     filesize = 0x3E + bitmapsize;
00274     fprintf (outfp, "%c", (unsigned char)((filesize >> 24) & 0xff));
00275     fprintf (outfp, "%c", (unsigned char)((filesize >> 16) & 0xff));
00276     fprintf (outfp, "%c", (unsigned char)((filesize >> 8) & 0xff));
00277     fprintf (outfp, "%c", (unsigned char)(filesize & 0xff));
00278
00279     /* Reserved - 0's */
00280     fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00281
00282     /* Offset from start of file to bitmap data */
00283     fprintf (outfp, "%c%c%c%c", 0x3E, 0x00, 0x00, 0x00);
00284
00285     /* Length of bitmap info header */
00286     fprintf (outfp, "%c%c%c%c", 0x28, 0x00, 0x00, 0x00);
00287
00288     /* Width of bitmap in pixels */
00289     fprintf (outfp, "%c%c%c%c", 0x40, 0x02, 0x00, 0x00);
00290
00291     /* Height of bitmap in pixels */
00292     fprintf (outfp, "%c%c%c%c", 0x20, 0x02, 0x00, 0x00);
00293
00294     /* Planes in bitmap (fixed at 1) */
00295     fprintf (outfp, "%c%c", 0x01, 0x00);
00296
00297     /* bits per pixel (1 = monochrome) */
00298     fprintf (outfp, "%c%c", 0x01, 0x00);
00299
00300     /* Compression (0 = none) */
00301     fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00302
00303     /* Size of bitmap data in bytes */
00304

```

```

00305     fprintf (outfp, "%c", (unsigned char)((bitmapsize      ) & 0xff));
00306     fprintf (outfp, "%c", (unsigned char)((bitmapsize » 0x08) & 0xff));
00307     fprintf (outfp, "%c", (unsigned char)((bitmapsize » 0x10) & 0xff));
00308     fprintf (outfp, "%c", (unsigned char)((bitmapsize » 0x18) & 0xff));
00309
00310     /* Horizontal resolution in pixels per meter */
00311     fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);
00312
00313     /* Vertical resolution in pixels per meter */
00314     fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);
00315
00316     /* Number of colors used */
00317     fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);
00318
00319     /* Number of important colors */
00320     fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);
00321
00322     /* The color black: B=0x00, G=0x00, R=0x00, Filler=0xFF */
00323     fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);
00324
00325     /* The color white: B=0xFF, G=0xFF, R=0xFF, Filler=0xFF */
00326     fprintf (outfp, "%c%c%c%c", 0xFF, 0xFF, 0xFF, 0x00);
00327
00328     /*
00329     Now write the raw data bits.  Data is written from the lower
00330     left-hand corner of the image to the upper right-hand corner
00331     of the image.
00332     */
00333     for (toppixelrow=17*32-1; toppixelrow >= 0; toppixelrow--) {
00334         for (j=0; j<18; j++) {
00335             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2)   ]);
00336             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 1]);
00337             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 2]);
00338
00339             fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 3]);
00340         }
00341     }
00342 }
00343 exit (0);
00344 }
00345
00346
00347 /**
00348  @brief Generate a bitmap for one glyph.
00349
00350  Convert the portion of a hex string after the ':' into a character bitmap.
00351
00352  If string is >= 128 characters, it will fill all 4 bytes per row.
00353  If string is >= 64 characters and < 128, it will fill 2 bytes per row.
00354  Otherwise, it will fill 1 byte per row.
00355
00356  @param[in] instring The character array containing the glyph bitmap.
00357  @param[out] character Glyph bitmap, 8, 16, or 32 columns by 16 rows tall.
00358  @return Always returns 0.
00359  */
00360 int
00361 hex2bit (char *instring, unsigned char character[32][4])
00362 {
00363
00364     int i; /* current row in bitmap character */
00365     int j; /* current character in input string */
00366     int k; /* current byte in bitmap character */
00367     int width; /* number of output bytes to fill - 1: 0, 1, 2, or 3 */
00368
00369     for (i=0; i<32; i++) /* erase previous character */
00370         character[i][0] = character[i][1] = character[i][2] = character[i][3] = 0;
00371     j=0; /* current location is at beginning of instring */
00372
00373     if (strlen (instring) <= 34) /* 32 + possible '\r', '\n' */
00374         width = 0;
00375     else if (strlen (instring) <= 66) /* 64 + possible '\r', '\n' */
00376         width = 1;
00377     else if (strlen (instring) <= 98) /* 96 + possible '\r', '\n' */
00378         width = 3;
00379     else /* the maximum allowed is quadruple-width */
00380         width = 4;
00381
00382     k = (width > 1) ? 0 : 1; /* if width > double, start at index 1 else at 0 */
00383
00384     for (i=8; i<24; i++) { /* 16 rows per input character, rows 8..23 */
00385         sscanf (&instring[j], "%2hhx", &character[i][k]);

```

```

00386     j += 2;
00387     if (width > 0) { /* add next pair of hex digits to this row */
00388         sscanf (&instring[j], "%2hhx", &character[i][k+1]);
00389         j += 2;
00390         if (width > 1) { /* add next pair of hex digits to this row */
00391             sscanf (&instring[j], "%2hhx", &character[i][k+2]);
00392             j += 2;
00393             if (width > 2) { /* quadruple-width is maximum width */
00394                 sscanf (&instring[j], "%2hhx", &character[i][k+3]);
00395                 j += 2;
00396             }
00397         }
00398     }
00399 }
00400
00401 return (0);
00402 }
00403
00404 /**
00405  * @brief Initialize the bitmap grid.
00406  * @param[out] bitmap The bitmap to generate, with 32x32 pixel glyph areas.
00407  * @return Always returns 0.
00408  */
00409
00410 int
00411 init (unsigned char bitmap[17*32][18*4])
00412 {
00413     int i, j;
00414     unsigned char charbits[32][4]; /* bitmap for one character, 4 bytes/row */
00415     unsigned toppixelrow;
00416     unsigned thiscol;
00417     unsigned char pnybble0, pnybble1, pnybble2, pnybble3;
00418
00419     for (i=0; i<18; i++) { /* bitmaps for '0'..'9', 'A'..'F', 'u', '+' */
00420         hex2bit (&hex[i][5], charbits); /* convert hex string to 32*4 bitmap */
00421
00422         for (j=0; j<32; j++) hexbits[i][j] = ~charbits[j][1];
00423     }
00424
00425     /*
00426      * Initialize bitmap to all white.
00427      */
00428     for (toppixelrow=0; toppixelrow < 17*32; toppixelrow++) {
00429         for (thiscol=0; thiscol<18; thiscol++) {
00430             bitmap[toppixelrow][(thiscol « 2) ] = 0xff;
00431             bitmap[toppixelrow][(thiscol « 2) | 1] = 0xff;
00432             bitmap[toppixelrow][(thiscol « 2) | 2] = 0xff;
00433             bitmap[toppixelrow][(thiscol « 2) | 3] = 0xff;
00434         }
00435     }
00436
00437     /*
00438      * Write the "u+nnnn" table header in the upper left-hand corner,
00439      * where nnnn is the upper 16 bits of a 32-bit Unicode assignment.
00440      */
00441     pnybble3 = (unipage » 20);
00442     pnybble2 = (unipage » 16) & 0xf;
00443     pnybble1 = (unipage » 12) & 0xf;
00444     pnybble0 = (unipage » 8) & 0xf;
00445     for (i=0; i<32; i++) {
00446         bitmap[i][1] = hexbits[16][i]; /* copy 'u' */
00447         bitmap[i][2] = hexbits[17][i]; /* copy '+' */
00448         bitmap[i][3] = hexbits[pnybble3][i];
00449         bitmap[i][4] = hexbits[pnybble2][i];
00450         bitmap[i][5] = hexbits[pnybble1][i];
00451         bitmap[i][6] = hexbits[pnybble0][i];
00452     }
00453
00454     /*
00455      * Write low-order 2 bytes of Unicode number assignments, as hex labels
00456      */
00457     pnybble3 = (unipage » 4) & 0xf; /* Highest-order hex digit */
00458     pnybble2 = (unipage » 0) & 0xf; /* Next highest-order hex digit */
00459
00460     /*
00461      * Write the column headers in bitmap[] (row headers if flipped)
00462      */
00463     toppixelrow = 32 * 17 - 1; /* maximum pixel row number */
00464
00465     /*
00466      * Label the column headers. The hexbits[] bytes are split across two
00467      * bitmap[] entries to center a the hex digits in a column of 4 bytes.
00468      * OR highest byte with 0xf0 and lowest byte with 0x0f to make outer

```

```

00467     nybbles white (0=black, 1-white).
00468     */
00469     for (i=0; i<16; i++) {
00470         for (j=0; j<32; j++) {
00471             if (!flip) { /* transpose matrix */
00472                 bitmap[j][((i+2) « 2) | 0] = (hexbits[pybble3][j] » 4) | 0xf0;
00473                 bitmap[j][((i+2) « 2) | 1] = (hexbits[pybble3][j] « 4) |
00474                     (hexbits[pybble2][j] » 4);
00475                 bitmap[j][((i+2) « 2) | 2] = (hexbits[pybble2][j] « 4) |
00476                     (hexbits[i][j] » 4);
00477                 bitmap[j][((i+2) « 2) | 3] = (hexbits[i][j] « 4) | 0x0f;
00478             }
00479             else {
00480                 bitmap[j][((i+2) « 2) | 1] = (hexbits[i][j] » 4) | 0xf0;
00481                 bitmap[j][((i+2) « 2) | 2] = (hexbits[i][j] « 4) | 0x0f;
00482             }
00483         }
00484     }
00485     /*
00486     Now use the single hex digit column graphics to label the row headers.
00487     */
00488     for (i=0; i<16; i++) {
00489         toppixelrow = 32 * (i + 1) - 1; /* from bottom to top */
00490
00491         for (j=0; j<32; j++) {
00492             if (!flip) { /* if not transposing matrix */
00493                 bitmap[toppixelrow + j][4] = hexbits[pybble3][j];
00494                 bitmap[toppixelrow + j][5] = hexbits[pybble2][j];
00495             }
00496             bitmap[toppixelrow + j][6] = hexbits[i][j];
00497         }
00498     }
00499     /*
00500     Now draw grid lines in bitmap, around characters we just copied.
00501     */
00502     /* draw vertical lines 2 pixels wide */
00503     for (i=1*32; i<17*32; i++) {
00504         if ((i & 0x1f) == 7)
00505             i++;
00506         else if ((i & 0x1f) == 14)
00507             i += 2;
00508         else if ((i & 0x1f) == 22)
00509             i++;
00510         for (j=1; j<18; j++) {
00511             bitmap[i][(j « 2) | 3] &= 0xfe;
00512         }
00513     }
00514     /* draw horizontal lines 1 pixel tall */
00515     for (i=1*32-1; i<18*32-1; i+=32) {
00516         for (j=2; j<18; j++) {
00517             bitmap[i][(j « 2) ] = 0x00;
00518             bitmap[i][(j « 2) | 1] = 0x81;
00519             bitmap[i][(j « 2) | 2] = 0x81;
00520             bitmap[i][(j « 2) | 3] = 0x00;
00521         }
00522     }
00523     /* fill in top left corner pixel of grid */
00524     bitmap[31][7] = 0xfe;
00525
00526     return (0);
00527 }

```

5.27 src/unihexgen.c File Reference

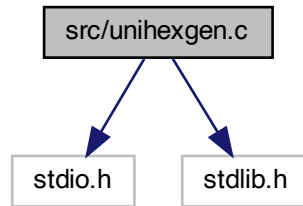
unihexgen - Generate a series of glyphs containing hexadecimal code points

```

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

```


Include dependency graph for unihexgen.c:



Functions

- int `main` (int argc, char *argv[])
The main function.
- void `hexprint4` (int thiscp)
Generate a bitmap containing a 4-digit Unicode code point.
- void `hexprint6` (int thiscp)
Generate a bitmap containing a 6-digit Unicode code point.

Variables

- char `hexdigit` [16][5]
Bitmap pattern for each hexadecimal digit.

5.27.1 Detailed Description

unihexgen - Generate a series of glyphs containing hexadecimal code points

Author

Paul Hardy

Copyright

Copyright (C) 2013 Paul Hardy

This program generates glyphs in Unifont .hex format that contain four- or six-digit hexadecimal numbers in a 16x16 pixel area. These are rendered as white digits on a black background.

argv[1] is the starting code point (as a hexadecimal string, with no leading "0x").

argv[2] is the ending code point (as a hexadecimal string, with no leading "0x").

For example:

```
unihexgen e000 f8ff > pua.hex
```

This generates the Private Use Area glyph file.

This utility program works in Roman Czyborra's unifont.hex file format, the basis of the GNU Unifont package.

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

5.27.2 Function Documentation

5.27.2.1 hexprint4()

```
void hexprint4 (
    int thiscp )
```

Generate a bitmap containing a 4-digit Unicode code point.

Takes a 4-digit Unicode code point as an argument and prints a unifont.hex string for it to stdout.

Parameters

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

Definition at line 160 of file [unihexgen.c](#).

```

00161 {
00162
00163     int grid[16]; /* the glyph grid we'll build */
00164
00165     int row;      /* row number in current glyph */
00166     int digitrow; /* row number in current hex digit being rendered */
00167     int rowbits;  /* 1 & 0 bits to draw current glyph row */
00168
00169     int d1, d2, d3, d4; /* four hexadecimal digits of each code point */
00170
00171     d1 = (thiscp » 12) & 0xF;
00172     d2 = (thiscp » 8) & 0xF;
00173     d3 = (thiscp » 4) & 0xF;
00174     d4 = (thiscp      ) & 0xF;
00175
00176     /* top and bottom rows are white */
00177     grid[0] = grid[15] = 0x0000;
00178
00179     /* 14 inner rows are 14-pixel wide black lines, centered */
00180     for (row = 1; row < 15; row++) grid[row] = 0x7FFE;
00181
00182     printf ("%04X:", thiscp);
00183
00184     /*
00185      * Render the first row of 2 hexadecimal digits
00186      */
00187     digitrow = 0; /* start at top of first row of digits to render */
00188     for (row = 2; row < 7; row++) {
00189         rowbits = (hexdigit[d1][digitrow] « 9) |
00190                 (hexdigit[d2][digitrow] « 3);
00191         grid[row] ^= rowbits; /* digits appear as white on black background */
00192         digitrow++;
00193     }
00194
00195     /*
00196      * Render the second row of 2 hexadecimal digits
00197      */
00198     digitrow = 0; /* start at top of first row of digits to render */
00199     for (row = 9; row < 14; row++) {
00200         rowbits = (hexdigit[d3][digitrow] « 9) |
00201                 (hexdigit[d4][digitrow] « 3);
00202         grid[row] ^= rowbits; /* digits appear as white on black background */
00203         digitrow++;
00204     }
00205
00206     for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);
00207
00208     putchar ('\n');
00209
00210     return;
00211 }

```

Here is the caller graph for this function:



5.27.2.2 hexprint6()

```

void hexprint6 (
    int thiscp )

```

Generate a bitmap containing a 6-digit Unicode code point.

Takes a 6-digit Unicode code point as an argument and prints a unifont.hex string for it to stdout.

Parameters

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

Definition at line 223 of file [unihexgen.c](#).

```

00224 {
00225
00226     int grid[16]; /* the glyph grid we'll build */
00227
00228     int row;      /* row number in current glyph */
00229     int digitrow; /* row number in current hex digit being rendered */
00230     int rowbits;  /* 1 & 0 bits to draw current glyph row */
00231
00232     int d1, d2, d3, d4, d5, d6; /* six hexadecimal digits of each code point */
00233
00234     d1 = (thiscp » 20) & 0xF;
00235     d2 = (thiscp » 16) & 0xF;
00236     d3 = (thiscp » 12) & 0xF;
00237     d4 = (thiscp » 8)  & 0xF;
00238     d5 = (thiscp » 4)  & 0xF;
00239     d6 = (thiscp      ) & 0xF;
00240
00241     /* top and bottom rows are white */
00242     grid[0] = grid[15] = 0x0000;
00243
00244     /* 14 inner rows are 16-pixel wide black lines, centered */
00245     for (row = 1; row < 15; row++) grid[row] = 0xFFFF;
00246
00247
00248     printf ("%06X:", thiscp);
00249
00250     /*
00251      * Render the first row of 3 hexadecimal digits
00252      */
00253     digitrow = 0; /* start at top of first row of digits to render */
00254     for (row = 2; row < 7; row++) {
00255         rowbits = (hexdigit[d1][digitrow] « 11) |
00256                 (hexdigit[d2][digitrow] « 6) |
00257                 (hexdigit[d3][digitrow] « 1);
00258         grid[row] ^= rowbits; /* digits appear as white on black background */
00259         digitrow++;
00260     }
00261
00262     /*
00263      * Render the second row of 3 hexadecimal digits
00264      */
00265     digitrow = 0; /* start at top of first row of digits to render */
00266     for (row = 9; row < 14; row++) {
00267         rowbits = (hexdigit[d4][digitrow] « 11) |
00268                 (hexdigit[d5][digitrow] « 6) |
00269                 (hexdigit[d6][digitrow] « 1);
00270         grid[row] ^= rowbits; /* digits appear as white on black background */
00271         digitrow++;
00272     }
00273
00274     for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);
00275

```

```
00276 putchar ('\n');
00277
00278 return;
00279 }
```

Here is the caller graph for this function:



5.27.2.3 main()

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

The main function.

Parameters

in	argc	The count of command line arguments.
in	argv	Pointer to array of command line arguments (code point range).

Returns

This program exits with status `EXIT_SUCCESS`.

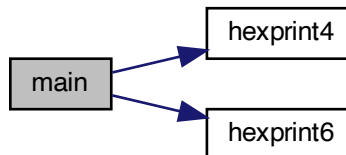
Definition at line 112 of file `unihexgen.c`.

```

00113 {
00114
00115     int startcp, endcp, thiscp;
00116     void hexprint4(int); /* function to print one 4-digit unifont.hex code point */
00117     void hexprint6(int); /* function to print one 6-digit unifont.hex code point */
00118
00119     if (argc != 3) {
00120         fprintf (stderr, "\n%s - generate unifont.hex code points as\n", argv[0]);
00121         fprintf (stderr, "four-digit hexadecimal numbers in a 2 by 2 grid,\n");
00122         fprintf (stderr, "or six-digit hexadecimal numbers in a 3 by 2 grid.\n");
00123         fprintf (stderr, "Syntax:\n\n");
00124         fprintf (stderr, "    %s first_code_point last_code_point > glyphs.hex\n\n", argv[0]);
00125         fprintf (stderr, "Example (to generate glyphs for the Private Use Area):\n\n");
00126         fprintf (stderr, "    %s e000 f8ff > pua.hex\n\n", argv[0]);
00127         exit (EXIT_FAILURE);
00128     }
00129
00130     sscanf (argv[1], "%x", &startcp);
00131     sscanf (argv[2], "%x", &endcp);
00132
00133     startcp &= 0xFFFFFFF; /* limit to 6 hex digits */
00134     endcp   &= 0xFFFFFFF; /* limit to 6 hex digits */
00135
00136     /*
00137     For each code point in the desired range, generate a glyph.
00138     */
00139     for (thiscp = startcp; thiscp <= endcp; thiscp++) {
00140         if (thiscp <= 0xFFFFF) {
00141             hexprint4 (thiscp); /* print digits 2/line, 2 lines */
00142         }
00143         else {
00144             hexprint6 (thiscp); /* print digits 3/line, 2 lines */
00145         }
00146     }
00147     exit (EXIT_SUCCESS);
00148 }

```

Here is the call graph for this function:



5.27.3 Variable Documentation

5.27.3.1 hexdigit

```
char hexdigit[16][5]
```

Initial value:

```
= {
    {0x6,0x9,0x9,0x9,0x6},
    {0x2,0x6,0x2,0x2,0x7},
    {0xF,0x1,0xF,0x8,0xF},
    {0xE,0x1,0x7,0x1,0xE},
    {0x9,0x9,0xF,0x1,0x1},
    {0xF,0x8,0xF,0x1,0xF},
    {0x6,0x8,0xE,0x9,0x6},
    {0xF,0x1,0x2,0x4,0x4},
    {0x6,0x9,0x6,0x9,0x6},
    {0x6,0x9,0x7,0x1,0x6},
    {0xF,0x9,0xF,0x9,0x9},
    {0xE,0x9,0xE,0x9,0xE},
    {0x7,0x8,0x8,0x8,0x7},
    {0xE,0x9,0x9,0x9,0xE},
    {0xF,0x8,0xE,0x8,0xF},
    {0xF,0x8,0xE,0x8,0x8}
}
```

Bitmap pattern for each hexadecimal digit.

hexdigit[][] definition: the bitmap pattern for each hexadecimal digit.

Each digit is drawn as a 4 wide by 5 high bitmap, so each digit row is one hexadecimal digit, and each entry has 5 rows.

For example, the entry for digit 1 is:

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

which corresponds graphically to:

```
-#- ==> 0010 ==> 0x2 -##- ==> 0110 ==> 0x6 -#- ==> 0010 ==> 0x2 -#- ==> 0010 ==> 0x2
-### ==> 0111 ==> 0x7
```

These row values will then be exclusive-ORed with four one bits (binary 1111, or 0xF) to form white digits on a black background.

Functions hexprint4 and hexprint6 share the hexdigit array; they print four-digit and six-digit hexadecimal code points in a single glyph, respectively.

Definition at line 84 of file [unihexgen.c](#).

5.28 unihexgen.c

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

```

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 #include <stdio.h>
00051 #include <stdlib.h>
00052
00053
00054 /**
00055  @brief Bitmap pattern for each hexadecimal digit.
00056
00057  hexdigit[][] definition: the bitmap pattern for
00058  each hexadecimal digit.
00059
00060  Each digit is drawn as a 4 wide by 5 high bitmap,
00061  so each digit row is one hexadecimal digit, and
00062  each entry has 5 rows.
00063
00064  For example, the entry for digit 1 is:
00065
00066      {0x2,0x6,0x2,0x2,0x7},
00067
00068  which corresponds graphically to:
00069
00070      --#- ==> 0010 ==> 0x2
00071      -##- ==> 0110 ==> 0x6
00072      --#- ==> 0010 ==> 0x2
00073      --#- ==> 0010 ==> 0x2
00074      -### ==> 0111 ==> 0x7
00075
00076  These row values will then be exclusive-ORed with four one bits

```



```

00077 (binary 1111, or 0xF) to form white digits on a black background.
00078
00079
00080 Functions hexprint4 and hexprint6 share the hexdigit array;
00081 they print four-digit and six-digit hexadecimal code points
00082 in a single glyph, respectively.
00083 */
00084 char hexdigit[16][5] = {
00085     {0x6,0x9,0x9,0x9,0x6}, /* 0x0 */
00086     {0x2,0x6,0x2,0x2,0x7}, /* 0x1 */
00087     {0xF,0x1,0xF,0x8,0xF}, /* 0x2 */
00088     {0xE,0x1,0x7,0x1,0xE}, /* 0x3 */
00089     {0x9,0x9,0xF,0x1,0x1}, /* 0x4 */
00090     {0xF,0x8,0xF,0x1,0xF}, /* 0x5 */
00091     {0x6,0x8,0xE,0x9,0x6}, /* 0x6 */ // {0x8,0x8,0xF,0x9,0xF} [alternate square form of 6]
00092     {0xF,0x1,0x2,0x4,0x4}, /* 0x7 */
00093     {0x6,0x9,0x6,0x9,0x6}, /* 0x8 */
00094     {0x6,0x9,0x7,0x1,0x6}, /* 0x9 */ // {0xF,0x9,0xF,0x1,0x1} [alternate square form of 9]
00095     {0xF,0x9,0xF,0x9,0x9}, /* 0xA */
00096     {0xE,0x9,0xE,0x9,0xE}, /* 0xB */
00097     {0x7,0x8,0x8,0x8,0x7}, /* 0xC */
00098     {0xE,0x9,0x9,0x9,0xE}, /* 0xD */
00099     {0xF,0x8,0xE,0x8,0xF}, /* 0xE */
00100     {0xF,0x8,0xE,0x8,0x8}, /* 0xF */
00101 };
00102
00103
00104 /**
00105  @brief The main function.
00106
00107  @param[in] argc The count of command line arguments.
00108  @param[in] argv Pointer to array of command line arguments (code point range).
00109  @return This program exits with status EXIT_SUCCESS.
00110 */
00111 int
00112 main (int argc, char *argv[])
00113 {
00114
00115     int startcp, endcp, thiscp;
00116     void hexprint4(int); /* function to print one 4-digit unifont.hex code point */
00117     void hexprint6(int); /* function to print one 6-digit unifont.hex code point */
00118
00119     if (argc != 3) {
00120         fprintf (stderr, "\n%s - generate unifont.hex code points as\n", argv[0]);
00121         fprintf (stderr, "four-digit hexadecimal numbers in a 2 by 2 grid.\n");
00122         fprintf (stderr, "or six-digit hexadecimal numbers in a 3 by 2 grid.\n");
00123         fprintf (stderr, "Syntax: \n\n");
00124         fprintf (stderr, "    %s first_code_point last_code_point > glyphs.hex\n\n", argv[0]);
00125         fprintf (stderr, "Example (to generate glyphs for the Private Use Area):\n\n");
00126         fprintf (stderr, "    %s e000 f8ff > pua.hex\n\n", argv[0]);
00127         exit (EXIT_FAILURE);
00128     }
00129
00130     sscanf (argv[1], "%x", &startcp);
00131     sscanf (argv[2], "%x", &endcp);
00132
00133     startcp &= 0xFFFFF; /* limit to 6 hex digits */
00134     endcp &= 0xFFFFF; /* limit to 6 hex digits */
00135
00136     /*
00137      For each code point in the desired range, generate a glyph.
00138     */
00139     for (thiscp = startcp; thiscp <= endcp; thiscp++) {
00140         if (thiscp <= 0xFFFF) {
00141             hexprint4 (thiscp); /* print digits 2/line, 2 lines */
00142         }
00143         else {
00144             hexprint6 (thiscp); /* print digits 3/line, 2 lines */
00145         }
00146     }
00147     exit (EXIT_SUCCESS);
00148 }
00149
00150
00151 /**
00152  @brief Generate a bitmap containing a 4-digit Unicode code point.
00153
00154  Takes a 4-digit Unicode code point as an argument
00155  and prints a unifont.hex string for it to stdout.
00156
00157  @param[in] thiscp The current code point for which to generate a glyph.

```

```

00158 */
00159 void
00160 hexprint4 (int thiscp)
00161 {
00162     int grid[16]; /* the glyph grid we'll build */
00163     int row;      /* row number in current glyph */
00164     int digitrow; /* row number in current hex digit being rendered */
00165     int rowbits;  /* 1 & 0 bits to draw current glyph row */
00166     int d1, d2, d3, d4; /* four hexadecimal digits of each code point */
00167     d1 = (thiscp » 12) & 0xF;
00168     d2 = (thiscp » 8) & 0xF;
00169     d3 = (thiscp » 4) & 0xF;
00170     d4 = (thiscp ) & 0xF;
00171     /* top and bottom rows are white */
00172     grid[0] = grid[15] = 0x0000;
00173     /* 14 inner rows are 14-pixel wide black lines, centered */
00174     for (row = 1; row < 15; row++) grid[row] = 0x7FFE;
00175     printf ("%04X:", thiscp);
00176     /*
00177      * Render the first row of 2 hexadecimal digits
00178      */
00179     digitrow = 0; /* start at top of first row of digits to render */
00180     for (row = 2; row < 7; row++) {
00181         rowbits = (hexdigit[d1][digitrow] « 9) |
00182                 (hexdigit[d2][digitrow] « 3);
00183         grid[row] ^= rowbits; /* digits appear as white on black background */
00184         digitrow++;
00185     }
00186     /*
00187      * Render the second row of 2 hexadecimal digits
00188      */
00189     digitrow = 0; /* start at top of first row of digits to render */
00190     for (row = 9; row < 14; row++) {
00191         rowbits = (hexdigit[d3][digitrow] « 9) |
00192                 (hexdigit[d4][digitrow] « 3);
00193         grid[row] ^= rowbits; /* digits appear as white on black background */
00194         digitrow++;
00195     }
00196     for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);
00197     putchar ('\n');
00198     return;
00199 }
00200 /**
00201  @brief Generate a bitmap containing a 6-digit Unicode code point.
00202  Takes a 6-digit Unicode code point as an argument
00203  and prints a unifont.hex string for it to stdout.
00204  @param[in] thiscp The current code point for which to generate a glyph.
00205  */
00206 void
00207 hexprint6 (int thiscp)
00208 {
00209     int grid[16]; /* the glyph grid we'll build */
00210     int row;      /* row number in current glyph */
00211     int digitrow; /* row number in current hex digit being rendered */
00212     int rowbits;  /* 1 & 0 bits to draw current glyph row */
00213     int d1, d2, d3, d4, d5, d6; /* six hexadecimal digits of each code point */
00214     d1 = (thiscp » 20) & 0xF;
00215     d2 = (thiscp » 16) & 0xF;
00216     d3 = (thiscp » 12) & 0xF;
00217     d4 = (thiscp » 8) & 0xF;
00218     d5 = (thiscp » 4) & 0xF;

```

```

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

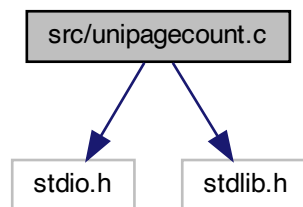
```

5.29 src/unipagecount.c File Reference

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

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

Include dependency graph for unipagecount.c:



Macros

- `#define MAXBUF 256`
Maximum input line size - 1.

Functions

- `int main (int argc, char *argv[])`
The main function.
- `void mkftable (unsigned plane, int pagecount[256], int links)`
Create an HTML table linked to PNG images.

5.29.1 Detailed Description

`unipagecount` - Count the number of glyphs defined in each page of 256 code points

Author

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

Copyright

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

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

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

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

Synopsis:

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

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

5.29.2 Macro Definition Documentation

5.29.2.1 MAXBUF

```
#define MAXBUF 256
```

Maximum input line size - 1.

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

5.29.3 Function Documentation

5.29.3.1 main()

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

The main function.

Parameters

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

Returns

This program exits with status 0.

Definition at line 67 of file [unipagecount.c](#).

```
00068 {  
00069  
00070     char inbuf[MAXBUF]; /* Max 256 characters in an input line */  
00071     int i, j; /* loop variables */  
00072     unsigned plane=0; /* Unicode plane number, 0 to 0x16 */  
00073     unsigned page; /* unicode page (256 bytes wide) */
```

```

00074 unsigned uchar; /* unicode character */
00075 int pagecount[256] = {256 * 0};
00076 int onepage=0; /* set to one if printing character grid for one page */
00077 int pageno=0; /* page number selected if only examining one page */
00078 int html=0; /* =0: print plain text; =1: print HTML */
00079 int links=0; /* =1: print HTML links; =0: don't print links */
00080 void mkftable(); /* make (print) flipped HTML table */
00081
00082 size_t strlen();
00083
00084 if (argc > 1 && argv[1][0] == '-') { /* Parse option */
00085     plane = 0;
00086     for (i = 1; i < argc; i++) {
00087         switch (argv[i][1]) {
00088             case 'p': /* specified -p<hexpage> -- use given page number */
00089                 sscanf (&argv[1][2], "%x", &pageno);
00090                 if (pageno >= 0 && pageno <= 255) onepage = 1;
00091                 break;
00092             case 'h': /* print HTML table instead of text table */
00093                 html = 1;
00094                 break;
00095             case 'l': /* print hyperlinks in HTML table */
00096                 links = 1;
00097                 html = 1;
00098                 break;
00099             case 'P': /* Plane number specified */
00100                 plane = atoi(&argv[1][2]);
00101                 break;
00102         }
00103     }
00104 }
00105 /*
00106     Initialize pagecount to account for noncharacters.
00107 */
00108 if (!onepage && plane==0) {
00109     pagecount[0xfd] = 32; /* for U+FDD0..U+FDEF */
00110 }
00111 pagecount[0xff] = 2; /* for U+nnFFFE, U+nnFFFF */
00112 /*
00113     Read one line at a time from input. The format is:
00114
00115     <hexpos>:<hexbitmap>
00116
00117     where <hexpos> is the hexadecimal Unicode character position
00118     in the range 00..FF and <hexbitmap> is the sequence of hexadecimal
00119     digits of the character, laid out in a grid from left to right,
00120     top to bottom. The character is assumed to be 16 rows of variable
00121     width.
00122 */
00123 while (fgets (inbuf, MAXBUF-1, stdin) != NULL) {
00124     sscanf (inbuf, "%X", &uchar);
00125     page = uchar » 8;
00126     if (onepage) { /* only increment counter if this is page we want */
00127         if (page == pageno) { /* character is in the page we want */
00128             pagecount[uchar & 0xff]++; /* mark character as covered */
00129         }
00130     }
00131     else { /* counting all characters in all pages */
00132         if (plane == 0) {
00133             /* Don't add in noncharacters (U+FDD0..U+FDEF, U+FFFE, U+FFFF) */
00134             if (uchar < 0xfdd0 || (uchar > 0xfdef && uchar < 0xfffe))
00135                 pagecount[page]++;
00136         }
00137         else {
00138             if ((page » 8) == plane) { /* code point is in desired plane */
00139                 pagecount[page & 0xFF]++;
00140             }
00141         }
00142     }
00143 }
00144 if (html) {
00145     mkftable (plane, pagecount, links);
00146 }
00147 else { /* Otherwise, print plain text table */
00148     if (plane > 0) fprintf (stdout, " ");
00149     fprintf (stdout,
00150             " 0 1 2 3 4 5 6 7 8 9 A B C D E F\n");
00151     for (i=0; i<0x10; i++) {
00152         fprintf (stdout, "%02X%X ", plane, i); /* row header */
00153         for (j=0; j<0x10; j++) {
00154             if (onepage) {

```

```
00155         if (pagecount[i*16+j])
00156             fprintf (stdout," * ");
00157         else
00158             fprintf (stdout," . ");
00159     }
00160     else {
00161         fprintf (stdout, "%3X ", pagecount[i*16+j]);
00162     }
00163 }
00164 fprintf (stdout, "\n");
00165 }
00166
00167 }
00168 exit (0);
00169 }
```

Here is the call graph for this function:



5.29.3.2 mkftable()

```
void mkftable (
    unsigned plane,
    int pagecount[256],
    int links )
```

Create an HTML table linked to PNG images.

This function creates an HTML table to show PNG files in a 16 by 16 grid. The background color of each "page" of 256 code points is shaded from red (for 0% coverage) to green (for 100% coverage).

Parameters

in	plane	The Uni-code plane, 0..17.
in	pagecount	Array with count of glyphs in each 256 code point range.
Generated by Doxygen		

Parameters

in	links	1 = generate hyperlinks, 0 = do not generate hyperlinks.
----	-------	--

Definition at line 185 of file [unipagecount.c](#).

```

00186 {
00187     int i, j;
00188     int count;
00189     unsigned bgcolor;
00190
00191     printf("<html>\n");
00192     printf("<body>\n");
00193     printf("<table border=\"3\" align=\"center\">\n");
00194     printf(" <tr><th colspan=\"16\" bgcolor=\"\#ffcc80\">");
00195     printf("GNU Unifont Glyphs<br>with Page Coverage for Plane %d<br>(Green=100%%, Red=0%%)</th></tr>\n",
plane);
00196     for (i = 0x0; i <= 0xF; i++) {
00197         printf(" <tr>\n");
00198         for (j = 0x0; j <= 0xF; j++) {
00199             count = pagecount[ (i « 4) | j ];
00200
00201             /* print link in cell if links == 1 */
00202             if (plane != 0 || (i < 0xd || (i == 0xd && j < 0x8) || (i == 0xf && j > 0x8))) {
00203                 /* background color is light green if completely done */
00204                 if (count == 0x100) bgcolor = 0xccffcc;
00205                 /* otherwise background is a shade of yellow to orange to red */
00206                 else bgcolor = 0xff0000 | (count « 8) | (count » 1);
00207                 printf(" <td bgcolor=\"%06X\">", bgcolor);
00208                 if (plane == 0)
00209                     printf(" <a href=\"png/plane%02X/uni%02X%X%X.png\">%X%X</a>", plane, plane, i, j, j);
00210                 else
00211                     printf(" <a href=\"png/plane%02X/uni%02X%X%X.png\">%02X%X%X</a>", plane, plane, i, j, plane, i, j);
00212                 printf(" </td>\n");
00213             }
00214             else if (i == 0xd) {
00215                 if (j == 0x8) {
00216                     printf(" <td align=\"center\" colspan=\"8\" bgcolor=\"\#cccccc\">");
00217                     printf("<b>Surrogate Pairs</b>");
00218                     printf(" </td>\n");
00219                 } /* otherwise don't print anything more columns in this row */
00220             }
00221             else if (i == 0xe) {
00222                 if (j == 0x0) {
00223                     printf(" <td align=\"center\" colspan=\"16\" bgcolor=\"\#cccccc\">");
00224                     printf("<b>Private Use Area</b>");
00225                     printf(" </td>\n");
00226                 } /* otherwise don't print any more columns in this row */
00227             }
00228             else if (i == 0xf) {
00229                 if (j == 0x0) {
00230                     printf(" <td align=\"center\" colspan=\"9\" bgcolor=\"\#cccccc\">");
00231                     printf("<b>Private Use Area</b>");
00232                     printf(" </td>\n");
00233                 }
00234             }
00235             }
00236         printf(" </tr>\n");
00237     }
00238     printf("</table>\n");
00239     printf("</body>\n");
00240     printf("</html>\n");
00241
00242     return;

```



```
00243 }
```

Here is the caller graph for this function:



5.30 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_file.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
```

```

00053 #include <stdio.h>
00054 #include <stdlib.h>
00055
00056 #define MAXBUF 256 ///< Maximum input line size - 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 0.
00065 */
00066 int
00067 main (int argc, char *argv[])
00068 {
00069     char inbuf[MAXBUF]; /* Max 256 characters in an input line */
00070     int i, j; /* loop variables */
00071     unsigned plane=0; /* Unicode plane number, 0 to 0x16 */
00072     unsigned page; /* unicode page (256 bytes wide) */
00073     unsigned unichar; /* unicode character */
00074     int pagecount[256] = {256 * 0};
00075     int onepage=0; /* set to one if printing character grid for one page */
00076     int pageno=0; /* page number selected if only examining one page */
00077     int html=0; /* =0: print plain text; =1: print HTML */
00078     int links=0; /* =1: print HTML links; =0: don't print links */
00079     void mkftable(); /* make (print) flipped HTML table */
00080     size_t strlen();
00081
00082     if (argc > 1 && argv[1][0] == '-') { /* Parse option */
00083         plane = 0;
00084         for (i = 1; i < argc; i++) {
00085             switch (argv[i][1]) {
00086                 case 'p': /* specified -p<hexpage> -- use given page number */
00087                     sscanf (&argv[i][2], "%x", &pageno);
00088                     if (pageno >= 0 && pageno <= 255) onepage = 1;
00089                     break;
00090                 case 'h': /* print HTML table instead of text table */
00091                     html = 1;
00092                     break;
00093                 case 'l': /* print hyperlinks in HTML table */
00094                     links = 1;
00095                     html = 1;
00096                     break;
00097                 case 'P': /* Plane number specified */
00098                     plane = atoi(&argv[i][2]);
00099                     break;
00100             }
00101         }
00102     }
00103     /*
00104     Initialize pagecount to account for noncharacters.
00105     */
00106     if (!onepage && plane==0) {
00107         pagecount[0xfd] = 32; /* for U+FDD0..U+FDEF */
00108     }
00109     pagecount[0xff] = 2; /* for U+nnFFFE, U+nnFFFF */
00110     /*
00111     Read one line at a time from input. The format is:
00112
00113         <hexpos>:<hexbitmap>
00114
00115     where <hexpos> is the hexadecimal Unicode character position
00116     in the range 00..FF and <hexbitmap> is the sequence of hexadecimal
00117     digits of the character, laid out in a grid from left to right,
00118     top to bottom. The character is assumed to be 16 rows of variable
00119     width.
00120     */
00121     while (fgets (inbuf, MAXBUF-1, stdin) != NULL) {
00122         sscanf (inbuf, "%X", &unichar);
00123         page = unichar » 8;
00124         if (onepage) { /* only increment counter if this is page we want */
00125             if (page == pageno) { /* character is in the page we want */
00126                 pagecount[unichar & 0xff]++; /* mark character as covered */
00127             }
00128         }
00129         else { /* counting all characters in all pages */
00130             if (plane == 0) {
00131                 /* Don't add in noncharacters (U+FDD0..U+FDEF, U+FFFE, U+FFFF) */

```

```

00134         if (unichar < 0xfdd0 || (unichar > 0xfdef && unichar < 0xfffe))
00135             pagecount[page]++;
00136     }
00137     else {
00138         if ((page » 8) == plane) { /* code point is in desired plane */
00139             pagecount[page & 0xFF]++;
00140         }
00141     }
00142 }
00143 }
00144 if (html) {
00145     mkftable (plane, pagecount, links);
00146 }
00147 else { /* Otherwise, print plain text table */
00148     if (plane > 0) fprintf (stdout, " ");
00149     fprintf (stdout,
00150             " 0 1 2 3 4 5 6 7 8 9 A B C D E F\n");
00151     for (i=0; i<0x10; i++) {
00152         fprintf (stdout, "%02X%X ", plane, i); /* row header */
00153         for (j=0; j<0x10; j++) {
00154             if (onepage) {
00155                 if (pagecount[i*16+j])
00156                     fprintf (stdout, " * ");
00157                 else
00158                     fprintf (stdout, " . ");
00159             }
00160             else {
00161                 fprintf (stdout, "%3X ", pagecount[i*16+j]);
00162             }
00163         }
00164         fprintf (stdout, "\n");
00165     }
00166 }
00167 }
00168 exit (0);
00169 }
00170
00171 /**
00172  @brief Create an HTML table linked to PNG images.
00173
00174  This function creates an HTML table to show PNG files
00175  in a 16 by 16 grid. The background color of each "page"
00176  of 256 code points is shaded from red (for 0% coverage)
00177  to green (for 100% coverage).
00178
00179  @param[in] plane The Unicode plane, 0..17.
00180  @param[in] pagecount Array with count of glyphs in each 256 code point range.
00181  @param[in] links 1 = generate hyperlinks, 0 = do not generate hyperlinks.
00182 */
00183 void
00184 mkftable (unsigned plane, int pagecount[256], int links)
00185 {
00186     int i, j;
00187     int count;
00188     unsigned bgcolor;
00189
00190     printf ("<html>\n");
00191     printf ("<body>\n");
00192     printf ("<table border=\t3\t align=\tcenter\t>\n");
00193     printf ("<tr><th colspan=\t16\t bgcolor=\t#ffcc80\t>");
00194     printf ("GNU Unifont Glyphs<br>with Page Coverage for Plane %d<br>(Green=100%%, Red=0%%)</th></tr>\n",
00195             plane);
00196     for (i = 0x0; i <= 0xF; i++) {
00197         printf ("<tr>\n");
00198         for (j = 0x0; j <= 0xF; j++) {
00199             count = pagecount[ (i « 4) | j ];
00200
00201             /* print link in cell if links == 1 */
00202             if (plane != 0 || (i < 0xd || (i == 0xd && j < 0x8) || (i == 0xf && j > 0x8))) {
00203                 /* background color is light green if completely done */
00204                 if (count == 0x100) bgcolor = 0xccffcc;
00205                 /* otherwise background is a shade of yellow to orange to red */
00206                 else bgcolor = 0xff0000 | (count « 8) | (count » 1);
00207                 printf ("<td bgcolor=\t\t#06X\t>", bgcolor);
00208                 if (plane == 0)
00209                     printf ("<a href=\tpng/plane%02X/uni%02X%X%X.png\t>%X%X</a>", plane, plane, i, j, j);
00210                 else
00211                     printf ("<a href=\tpng/plane%02X/uni%02X%X%X.png\t>%02X%X%X</a>", plane, plane, i, j, plane, i, j);
00212                 printf ("</td>\n");
00213             }

```

```
00214     else if (i == 0xd) {
00215         if (j == 0x8) {
00216             printf ("    <td align=\"center\" colspan=\"8\" bgcolor=\"#cccccc\">");
00217             printf ("<b>Surrogate Pairs</b>");
00218             printf ("</td>\n");
00219         } /* otherwise don't print anything more columns in this row */
00220     }
00221     else if (i == 0xe) {
00222         if (j == 0x0) {
00223             printf ("    <td align=\"center\" colspan=\"16\" bgcolor=\"#cccccc\">");
00224             printf ("<b>Private Use Area</b>");
00225             printf ("</td>\n");
00226         } /* otherwise don't print any more columns in this row */
00227     }
00228     else if (i == 0xf) {
00229         if (j == 0x0) {
00230             printf ("    <td align=\"center\" colspan=\"9\" bgcolor=\"#cccccc\">");
00231             printf ("<b>Private Use Area</b>");
00232             printf ("</td>\n");
00233         }
00234     }
00235 }
00236 printf (" </tr>\n");
00237 }
00238 printf ("</table>\n");
00239 printf ("</body>\n");
00240 printf ("</html>\n");
00241
00242 return;
00243 }
```

Index

- add_double_circle
 - unigencircles.c, [256](#)
- add_single_circle
 - unigencircles.c, [258](#)
- addByte
 - hex2otf.c, [32](#)
- addTable
 - hex2otf.c, [40](#)
- allBuffers
 - hex2otf.c, [117](#)
- ASCENDER
 - hex2otf.c, [32](#)
- ascii_bits
 - unifontpic.h, [251](#)
- ascii_hex
 - unifontpic.h, [251](#)
- B0
 - hex2otf.c, [32](#)
- B1
 - hex2otf.c, [33](#)
- begin
 - Buffer, [15](#)
- bitmap
 - Glyph, [18](#)
 - Options, [21](#)
- bits_per_pixel
 - unibmp2hex.c, [170](#)
- blankOutline
 - Options, [21](#)
- bmp_header
 - unibmp2hex.c, [171](#)
- Buffer, [15](#)
 - begin, [15](#)
 - capacity, [15](#)
 - end, [16](#)
 - hex2otf.c, [36](#)
 - next, [16](#)
- bufferCount
 - hex2otf.c, [117](#)
- buildOutline
 - hex2otf.c, [42](#)
- BX
 - hex2otf.c, [33](#)
- byCodePoint
 - hex2otf.c, [45](#)
- byTableTag
 - hex2otf.c, [46](#)
- byte
 - hex2otf.c, [37](#)
- byteCount
 - Glyph, [19](#)
- cacheBuffer
 - hex2otf.c, [46](#)
- cacheBytes
 - hex2otf.c, [47](#)
- cacheCFFOperand
 - hex2otf.c, [49](#)
- cacheStringAsUTF16BE
 - hex2otf.c, [50](#)
- cacheU16
 - hex2otf.c, [52](#)
- cacheU32
 - hex2otf.c, [53](#)
- cacheU8
 - hex2otf.c, [55](#)
- cacheZeros
 - hex2otf.c, [56](#)
- capacity
 - Buffer, [15](#)
- cff
 - Options, [22](#)
- checksum
 - TableRecord, [25](#)
- cleanBuffers
 - hex2otf.c, [57](#)
- codePoint
 - Glyph, [19](#)
- color_table
 - unibmp2hex.c, [171](#)
- combining
 - Glyph, [19](#)
- compression
 - unibmp2hex.c, [171](#)
- content
 - Table, [24](#)
- ContourOp
 - hex2otf.c, [38](#)
- DEFAULT_ID0

- hex2otf.h, 152
- DEFAULT_ID1
 - hex2otf.h, 152
- DEFAULT_ID11
 - hex2otf.h, 153
- DEFAULT_ID13
 - hex2otf.h, 153
- DEFAULT_ID14
 - hex2otf.h, 153
- DEFAULT_ID2
 - hex2otf.h, 153
- DEFAULT_ID5
 - hex2otf.h, 153
- defaultNames
 - hex2otf.h, 154
- defineStore
 - hex2otf.c, 33, 58
- DESCENDER
 - hex2otf.c, 33
- end
 - Buffer, 16
- ensureBuffer
 - hex2otf.c, 58
- fail
 - hex2otf.c, 60
- file_size
 - unibmp2hex.c, 171
- filetype
 - unibmp2hex.c, 171
- FILL_LEFT
 - hex2otf.c, 39
- FILL_RIGHT
 - hex2otf.c, 39
- fillBitmap
 - hex2otf.c, 62
- fillBlankOutline
 - hex2otf.c, 64
- fillCFF
 - hex2otf.c, 65
- fillCmapTable
 - hex2otf.c, 70
- fillGposTable
 - hex2otf.c, 72
- fillGsubTable
 - hex2otf.c, 73
- fillHeadTable
 - hex2otf.c, 75
- fillHheaTable
 - hex2otf.c, 77
- fillHmtxTable
 - hex2otf.c, 79
- fillMaxpTable
 - hex2otf.c, 80
- fillNameTable
 - hex2otf.c, 82
- fillOS2Table
 - hex2otf.c, 84
- fillPostTable
 - hex2otf.c, 86
- FillSide
 - hex2otf.c, 39
- fillTrueType
 - hex2otf.c, 88
- flip
 - unibmp2hex.c, 171
 - unihex2bmp.c, 286
- Font, 16
 - glyphCount, 17
 - glyphs, 17
 - maxWidth, 17
 - tables, 17
- forcewide
 - unibmp2hex.c, 172
- freeBuffer
 - hex2otf.c, 90
- FU
 - hex2otf.c, 34
- FUPEM
 - hex2otf.c, 34
- genlongbmp
 - unifontpic.c, 222
- genwidebmp
 - unifontpic.c, 227
- get_bytes
 - unibmpbump.c, 185
- gethex
 - unifontpic.c, 232
- Glyph, 18
 - bitmap, 18
 - byteCount, 19
 - codePoint, 19
 - combining, 19
 - hex2otf.c, 37
 - lsb, 19
 - pos, 19
- GLYPH_HEIGHT
 - hex2otf.c, 34
- GLYPH_MAX_BYTE_COUNT
 - hex2otf.c, 34
- GLYPH_MAX_WIDTH
 - hex2otf.c, 34
- glyphCount
 - Font, 17
- glyphs
 - Font, 17
- gpos

- Options, [22](#)
- gsub
 - Options, [22](#)
- HDR_LEN
 - unifontpic.c, [222](#)
- HEADER_STRING
 - unifontpic.h, [251](#)
- height
 - unibmp2hex.c, [172](#)
- hex
 - Options, [22](#)
 - unihex2bmp.c, [287](#)
- hex2bit
 - unihex2bmp.c, [278](#)
- hex2otf.c
 - addByte, [32](#)
 - addTable, [40](#)
 - allBuffers, [117](#)
 - ASCENDER, [32](#)
 - B0, [32](#)
 - B1, [33](#)
 - Buffer, [36](#)
 - bufferCount, [117](#)
 - buildOutline, [42](#)
 - BX, [33](#)
 - byCodePoint, [45](#)
 - byTableTag, [46](#)
 - byte, [37](#)
 - cacheBuffer, [46](#)
 - cacheBytes, [47](#)
 - cacheCFFOperand, [49](#)
 - cacheStringAsUTF16BE, [50](#)
 - cacheU16, [52](#)
 - cacheU32, [53](#)
 - cacheU8, [55](#)
 - cacheZeros, [56](#)
 - cleanBuffers, [57](#)
 - ContourOp, [38](#)
 - defineStore, [33](#), [58](#)
 - DESCENDER, [33](#)
 - ensureBuffer, [58](#)
 - fail, [60](#)
 - FILL_LEFT, [39](#)
 - FILL_RIGHT, [39](#)
 - fillBitmap, [62](#)
 - fillBlankOutline, [64](#)
 - fillCFF, [65](#)
 - fillCmapTable, [70](#)
 - fillGposTable, [72](#)
 - fillGsubTable, [73](#)
 - fillHeadTable, [75](#)
 - fillHheaTable, [77](#)
 - fillHmtxTable, [79](#)
 - fillMaxpTable, [80](#)
 - fillNameTable, [82](#)
 - fillOS2Table, [84](#)
 - fillPostTable, [86](#)
 - FillSide, [39](#)
 - fillTrueType, [88](#)
 - freeBuffer, [90](#)
 - FU, [34](#)
 - FUPEM, [34](#)
 - Glyph, [37](#)
 - GLYPH_HEIGHT, [34](#)
 - GLYPH_MAX_BYTE_COUNT, [34](#)
 - GLYPH_MAX_WIDTH, [34](#)
 - initBuffers, [91](#)
 - LOCA_OFFSET16, [40](#)
 - LOCA_OFFSET32, [40](#)
 - LocaFormat, [39](#)
 - main, [92](#)
 - matchToken, [94](#)
 - MAX_GLYPHS, [35](#)
 - MAX_NAME_IDS, [35](#)
 - NameStrings, [37](#)
 - newBuffer, [95](#)
 - nextBufferIndex, [117](#)
 - OP_CLOSE, [38](#)
 - OP_POINT, [38](#)
 - Options, [37](#)
 - organizeTables, [98](#)
 - parseOptions, [99](#)
 - pixels_t, [37](#)
 - positionGlyphs, [101](#)
 - prepareOffsets, [103](#)
 - prepareStringIndex, [104](#)
 - PRI_CP, [35](#)
 - printHelp, [105](#)
 - printVersion, [106](#)
 - PW, [35](#)
 - readCodePoint, [107](#)
 - readGlyphs, [108](#)
 - sortGlyphs, [110](#)
 - static_assert, [35](#)
 - Table, [38](#)
 - U16MAX, [36](#)
 - U32MAX, [36](#)
 - VERSION, [36](#)
 - writeBytes, [111](#)
 - writeFont, [112](#)
 - writeU16, [115](#)
 - writeU32, [116](#)
- hex2otf.h
 - DEFAULT_ID0, [152](#)
 - DEFAULT_ID1, [152](#)
 - DEFAULT_ID11, [153](#)
 - DEFAULT_ID13, [153](#)

- DEFAULT_ID14, [153](#)
 - DEFAULT_ID2, [153](#)
 - DEFAULT_ID5, [153](#)
 - defaultNames, [154](#)
 - NAMEPAIR, [154](#)
 - UNIFONT_VERSION, [154](#)
- hexbits
 - unihex2bmp.c, [287](#)
- hexdigit
 - unibmp2hex.c, [172](#)
 - unifontpic.h, [251](#)
 - unihexgen.c, [300](#)
- hexprint4
 - unihexgen.c, [296](#)
- hexprint6
 - unihexgen.c, [297](#)
- id
 - NamePair, [20](#)
- image_offset
 - unibmp2hex.c, [172](#)
- image_size
 - unibmp2hex.c, [172](#)
- important_colors
 - unibmp2hex.c, [173](#)
- info_size
 - unibmp2hex.c, [173](#)
- init
 - unihex2bmp.c, [280](#)
- initBuffers
 - hex2otf.c, [91](#)
- length
 - TableRecord, [25](#)
- LOCA_OFFSET16
 - hex2otf.c, [40](#)
- LOCA_OFFSET32
 - hex2otf.c, [40](#)
- LocaFormat
 - hex2otf.c, [39](#)
- lsb
 - Glyph, [19](#)
- main
 - hex2otf.c, [92](#)
 - unibdf2hex.c, [158](#)
 - unibmp2hex.c, [163](#)
 - unibmpbump.c, [186](#)
 - unicoverage.c, [202](#)
 - unidup.c, [213](#)
 - unifont1per.c, [217](#)
 - unifontpic.c, [234](#)
 - unigencircles.c, [259](#)
 - unigenwidth.c, [267](#)
 - unihex2bmp.c, [282](#)
 - unihexgen.c, [299](#)
 - unipagecount.c, [307](#)
- matchToken
 - hex2otf.c, [94](#)
- MAX_COMPRESSION_METHOD
 - unibmpbump.c, [185](#)
- MAX_GLYPHS
 - hex2otf.c, [35](#)
- MAX_NAME_IDS
 - hex2otf.c, [35](#)
- MAXBUF
 - unibdf2hex.c, [157](#)
 - unibmp2hex.c, [162](#)
 - unicoverage.c, [202](#)
 - unidup.c, [213](#)
 - unihex2bmp.c, [278](#)
 - unipagecount.c, [306](#)
- MAXFILENAME
 - unifont1per.c, [216](#)
- MAXSTRING
 - unifont1per.c, [216](#)
 - unifontpic.h, [251](#)
 - unigencircles.c, [256](#)
 - unigenwidth.c, [267](#)
- maxWidth
 - Font, [17](#)
- mkftable
 - unipagecount.c, [309](#)
- NAMEPAIR
 - hex2otf.h, [154](#)
- NamePair, [20](#)
 - id, [20](#)
 - str, [20](#)
- NameStrings
 - hex2otf.c, [37](#)
- nameStrings
 - Options, [22](#)
- ncolors
 - unibmp2hex.c, [173](#)
- newBuffer
 - hex2otf.c, [95](#)
- next
 - Buffer, [16](#)
- nextBufferIndex
 - hex2otf.c, [117](#)
- nextrange
 - unicoverage.c, [205](#)
- nplanes
 - unibmp2hex.c, [173](#)
- offset
 - TableRecord, [25](#)
- OP_CLOSE
 - hex2otf.c, [38](#)

- OP_POINT
 - hex2otf.c, 38
- Options, 21
 - bitmap, 21
 - blankOutline, 21
 - cff, 22
 - gpos, 22
 - gsub, 22
 - hex, 22
 - hex2otf.c, 37
 - nameStrings, 22
 - out, 22
 - pos, 23
 - truetype, 23
- organizeTables
 - hex2otf.c, 98
- out
 - Options, 22
- output2
 - unifontpic.c, 236
- output4
 - unifontpic.c, 237
- parseOptions
 - hex2otf.c, 99
- PIKTO_END
 - unigenwidth.c, 267
- PIKTO_SIZE
 - unigenwidth.c, 267
- PIKTO_START
 - unigenwidth.c, 267
- pixels_t
 - hex2otf.c, 37
- planeset
 - unibmp2hex.c, 173
- pos
 - Glyph, 19
 - Options, 23
- positionGlyphs
 - hex2otf.c, 101
- prepareOffsets
 - hex2otf.c, 103
- prepareStringIndex
 - hex2otf.c, 104
- PRI_CP
 - hex2otf.c, 35
- print_subtotal
 - unicoverage.c, 207
- printHelp
 - hex2otf.c, 105
- printVersion
 - hex2otf.c, 106
- PW
 - hex2otf.c, 35
- readCodePoint
 - hex2otf.c, 107
- readGlyphs
 - hex2otf.c, 108
- regrid
 - unibmpbump.c, 192
- sortGlyphs
 - hex2otf.c, 110
- src/hex2otf.c, 27, 118
- src/hex2otf.h, 150, 155
- src/unibdf2hex.c, 156, 159
- src/unibmp2hex.c, 161, 175
- src/unibmpbump.c, 184, 194
- src/unicoverage.c, 201, 208
- src/unidup.c, 211, 214
- src/unifont1per.c, 215, 218
- src/unifontpic.c, 221, 238
- src/unifontpic.h, 249, 252
- src/unigencircles.c, 255, 261
- src/unigenwidth.c, 265, 272
- src/unihex2bmp.c, 276, 288
- src/unihexgen.c, 294, 302
- src/unipagecount.c, 305, 311
- static_assert
 - hex2otf.c, 35
- str
 - NamePair, 20
- Table, 23
 - content, 24
 - hex2otf.c, 38
 - tag, 24
- TableRecord, 24
 - checksum, 25
 - length, 25
 - offset, 25
 - tag, 25
- tables
 - Font, 17
- tag
 - Table, 24
 - TableRecord, 25
- truetype
 - Options, 23
- U16MAX
 - hex2otf.c, 36
- U32MAX
 - hex2otf.c, 36
- unibdf2hex.c
 - main, 158
 - MAXBUF, 157
 - UNISTART, 157
 - UNISTOP, 158

- unibmp2hex.c
 - bits_per_pixel, 170
 - bmp_header, 171
 - color_table, 171
 - compression, 171
 - file_size, 171
 - filetype, 171
 - flip, 171
 - forcewide, 172
 - height, 172
 - hexdigit, 172
 - image_offset, 172
 - image_size, 172
 - important_colors, 173
 - info_size, 173
 - main, 163
 - MAXBUF, 162
 - ncolors, 173
 - nplanes, 173
 - planeset, 173
 - unidigit, 173
 - uniplane, 174
 - width, 174
 - x_ppm, 174
 - y_ppm, 174
- unibmpbump.c
 - get_bytes, 185
 - main, 186
 - MAX_COMPRESSION_METHOD, 185
 - regrid, 192
 - VERSION, 185
- unicoverage.c
 - main, 202
 - MAXBUF, 202
 - nextrange, 205
 - print_subtotal, 207
- unidigit
 - unibmp2hex.c, 173
- unidup.c
 - main, 213
 - MAXBUF, 213
- unifont1per.c
 - main, 217
 - MAXFILENAME, 216
 - MAXSTRING, 216
- UNIFONT_VERSION
 - hex2otf.h, 154
- unifontpic.c
 - genlongbmp, 222
 - genwidebmp, 227
 - gethex, 232
 - HDR_LEN, 222
 - main, 234
 - output2, 236
 - output4, 237
- unifontpic.h
 - ascii_bits, 251
 - ascii_hex, 251
 - HEADER_STRING, 251
 - hexdigit, 251
 - MAXSTRING, 251
- unigencircles.c
 - add_double_circle, 256
 - add_single_circle, 258
 - main, 259
 - MAXSTRING, 256
- unigenwidth.c
 - main, 267
 - MAXSTRING, 267
 - PIKTO_END, 267
 - PIKTO_SIZE, 267
 - PIKTO_START, 267
- unihex2bmp.c
 - flip, 286
 - hex, 287
 - hex2bit, 278
 - hexbits, 287
 - init, 280
 - main, 282
 - MAXBUF, 278
 - unipage, 287
- unihexgen.c
 - hexdigit, 300
 - hexprint4, 296
 - hexprint6, 297
 - main, 299
- unipage
 - unihex2bmp.c, 287
- unipagecount.c
 - main, 307
 - MAXBUF, 306
 - mkftable, 309
- uniplane
 - unibmp2hex.c, 174
- UNISTART
 - unibdf2hex.c, 157
- UNISTOP
 - unibdf2hex.c, 158
- VERSION
 - hex2otf.c, 36
 - unibmpbump.c, 185
- width
 - unibmp2hex.c, 174
- writeBytes
 - hex2otf.c, 111
- writeFont
 - hex2otf.c, 112

writeU16
 hex2otf.c, [115](#)
writeU32
 hex2otf.c, [116](#)

x_ppm
 unibmp2hex.c, [174](#)

y_ppm
 unibmp2hex.c, [174](#)