ttfautohint

Werner Lemberg

Version 0.93

Contents

1 Introduction

ttfautohint is a library written in C which takes a TrueType font as the input, removes its bytecode instructions (if any), and returns a new font where all glyphs are bytecode hinted using the information given by FreeType's autohinting module. The idea is to provide the excellent quality of the autohinter on platforms which don't use FreeType.

The library has a single API function, `TTF_autohint`, which is described below (chapter 4).

Bundled with the library there are two front-end programs, `ttfautohint` and `ttfautohintGUI` (chapter 2), being a command line and an application with a Graphics User Interface (GUI), respectively.

1.1 What exactly are hints?

To cite Wikipedia:

> ***Font hinting*** *(also known as* ***instructing****) is the use of mathematical instructions to adjust the display of an outline font so that it lines up with a rasterized grid. At low screen resolutions, hinting is critical for producing a clear, legible text. It can be accompanied by antialiasing and (on liquid crystal displays) subpixel rendering for further clarity.*

and Apple's TrueType Reference Manual:

> *For optimal results, a font instructor should follow these guidelines:*
>
> - *At small sizes, chance effects should not be allowed to magnify small differences in the original outline design of a glyph.*
> - *At large sizes, the subtlety of the original design should emerge.*

In general, there are three possible ways to hint a glyph.

1. The font contains hints (in the original sense of this word) to guide the rasterizer, telling it which shapes of the glyphs need special consideration. The hinting logic is partly in the font and partly in the rasterizer. More sophisticated rasterizers are able to produce better rendering results.

   This is how Type 1 and CFF font hints work.

2. The font contains exact instructions (also called *bytecode*) on how to move the points of its outlines, depending on the resolution of the output device, and which intentionally distort the (outline) shape to produce a well-rasterized result. The hinting logic is in the font; ideally, all rasterizers simply process these instructions to get the same result on all platforms.

   This is how TrueType hints work.

3. The font gets auto-hinted (at run-time). The hinting logic is completely in the rasterizer. No hints in the font are used or needed; instead, the rasterizer scans and analyzes the glyphs to apply corrections by itself.

   This is how FreeType's auto-hinter works; see below (chapter 3) for more.

1.2 What problems can arise with TrueType hinting?

While it is relatively easy to specify PostScript hints (either manually or by an auto-hinter which works at font creation time), creating TrueType hints is far more difficult. There are at least two reasons:

- TrueType instructions form a programming language, operating at a very low level. They are comparable to assembler code, thus lacking all high-level concepts to make programming more comfortable.

  Here an example how such code looks like:

  ```
  SVTCA[0]
  PUSHB[ ]  /* 3 values pushed */
  18 1 0
  CALL[ ]
  PUSHB[ ]  /* 2 values pushed */
  15 4
  MIRP[01001]
  PUSHB[ ]  /* 3 values pushed */
  7 3 0
  CALL[ ]
  ```

  Another major obstacle is the fact that font designers usually aren't programmers.

- It is very time consuming to manually hint glyphs. Given that the number of specialists for TrueType hinting is very limited, hinting a large set of glyphs for a font or font family can become very expensive.

1.3 Why ttfautohint?

The ttfautohint library brings the excellent quality of FreeType rendering to platforms which don't use FreeType, yet require hinting for text to look good – like Microsoft Windows. Roughly speaking, it converts the glyph analysis done by FreeType's auto-hinting module to TrueType bytecode. Internally, the auto-hinter's algorithm resembles PostScript hinting methods; it thus combines all three hinting methods discussed previously (section 1.1).

The simple interface of the front-ends (both on the command line and with the GUI) allows quick hinting of a whole font with a few mouse clicks or a single command on the prompt. As a result, you get better rendering results with web browsers, for example.

Across Windows rendering environments today, fonts processed with ttfautohint look best with ClearType enabled. This is the default for Windows 7. Good visual results are also seen in recent MacOS X versions and GNU/Linux systems that use FreeType for rendering.

The goal of the project is to generate a ‘first pass’ of hinting that font developers can refine further for ultimate quality.

2 ttfautohint and ttfautohintGUI

On all supported platforms (GNU/Linux, Windows, and Mac OS X), the GUI looks quite similar; the used toolkit is Qt, which in turn uses the platform's native widgets.

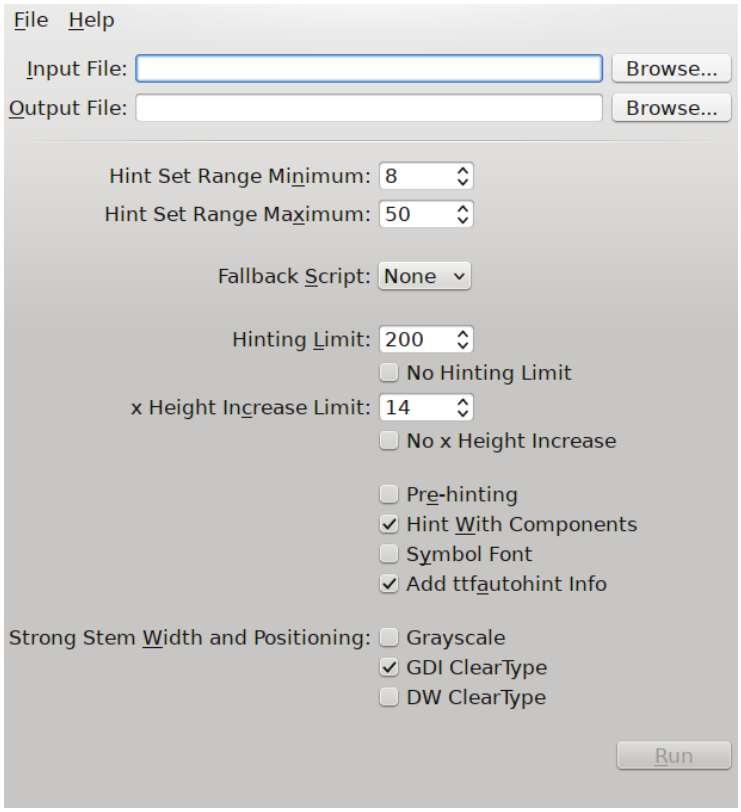

Figure 2.1: `ttfautohintGUI` on GNU/Linux running KDE

Both the GUI and console version share the same features, to be discussed in the next subsection.

Warning: ttfautohint cannot always process a font a second time. If the font contains composite glyphs, and option `-c` is not used, reprocessing with ttfautohint will fail. For this reason it is strongly recommended to *not* delete the original, unhinted font so that you can always rerun ttfautohint.

2.1 Calling `ttfautohint`

```
ttfautohint [OPTION]... [IN-FILE [OUT-FILE]]
```

The TTY binary, `ttfautohint`, works like a Unix filter, this is, it reads data from standard input if no input file name is given, and it sends its output to standard output if no output file name is specified.

A typical call looks like the following.

```
ttfautohint -v -f foo.ttf foo-autohinted.ttf
```

For demonstration purposes, here the same using a pipe and redirection. Note that Windows's default command line interpreter, `cmd.exe`, doesn't support piping with binary files, unfortunately.

```
cat foo.ttf | ttfautohint -v -f > foo-autohinted.ttf
```

2.2 Calling `ttfautohintGUI`

```
ttfautohintGUI [OPTION]...
```

`ttfautohintGUI` doesn't send any output to a console; however, it accepts the same command line options as `ttfautohint`, setting default values for the GUI.

2.3 Options

Long options can be given with one or two dashes, and with and without an equal sign between option and argument. This means that the following forms are acceptable: `-foo=`*bar*, `--foo=`*bar*, `-foo` *bar*, and `--foo` *bar*.

Below, the section title refers to the command's label in the GUI, then comes the name of the corresponding long command line option and its short equivalent, followed by a description.

Background and technical details on the meaning of the various options are given afterwards (chapter 3).

2.3.1 Hint Set Range Minimum, Hint Set Range Maximum

See 'Hint Sets' (section 3.5) for a definition and explanation.

`--hinting-range-min=`*n*, `-l` *n*
: The minimum PPEM value (in pixels) at which hint sets are created. The default value for *n* is 8.

`--hinting-range-max=`*n*, `-r` *n*
: The maximum PPEM value (in pixels) at which hint sets are created. The default value for *n* is 50.

2.3.2 Fallback Script

`--latin-fallback`, `-f`
: Set fallback script to 'latin', this is, use the 'latin' auto-hinting module instead of 'none' for all glyphs which can't be assigned to a script. See below (section 3.7) for more details.

2.3.3 Hinting Limit

`--hinting-limit=`*n*, `-G` *n*

The *hinting limit* is the PPEM value (in pixels) where hinting gets switched off (using the `INSTCTRL` bytecode instruction); it has zero impact on the file size. The default value for *n* is 200 which means that the font is not hinted for PPEM values larger than 200.

Note that hinting in the range 'hinting-range-max' up to 'hinting-limit' uses the hinting configuration for 'hinting-range-max'.

To omit a hinting limit, use `--hinting-limit=0` (or check the 'No Hinting Limit' box in the GUI). Since this will cause internal math overflow in the rasterizer for large pixel values (> 1500px approx.) it is strongly recommended to not use this except for testing purposes.

2.3.4 x Height Increase Limit

`--increase-x-height=`*n*, `-x` *n*

Normally, ttfautohint rounds the x height to the pixel grid, with a slight preference for rounding up. If this flag is set, values in the range 6 PPEM to *n* PPEM are much more often rounded up. The default value for *n* is 14. Use this flag to increase the legibility of small sizes if necessary; you might get weird rendering results otherwise for glyphs like 'a' or 'e', depending on the font design.

To switch off this feature, use `--increase-x-height=0` (or check the 'No x Height Increase' box in the GUI).

The following images again use the font 'Mertz Bold'.

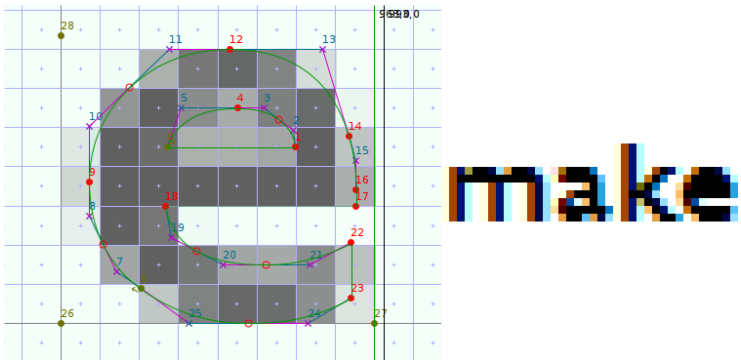

Figure 2.2: At 17px, without option `-x` and '`-w ""`', the hole in glyph 'e' looks very grey in the FontForge snapshot, and the GDI ClearType rendering (which is the default on older Windows versions) fills it completely with black because it uses B/W rendering along the y axis. FreeType's 'light' autohint mode (which corresponds to ttfautohint's 'smooth' stem width algorithm) intentionally aligns horizontal lines to non-integer (but still discrete) values to avoid large glyph shape distortions.

2.3.5 Pre-Hinting

`--pre-hinting`, `-p`

Pre-hinting means that a font's original bytecode is applied to all glyphs before it is replaced with bytecode created by ttfautohint. This makes only sense if your font already has some hints in it which modify the shape even at EM size (normally 2048px); for example, some CJK fonts need this

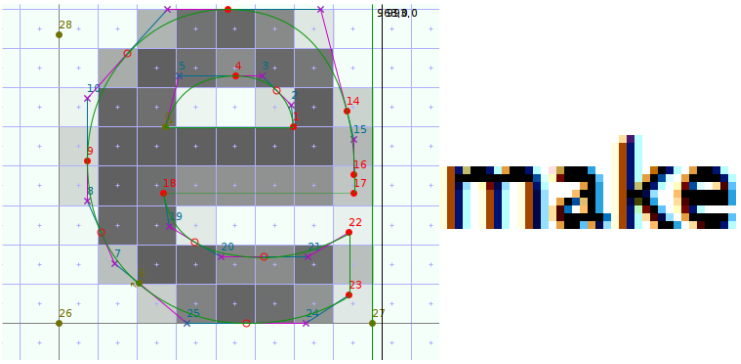

Figure 2.3: The same, this time with option `-x 17` (and '`-w ""`').

because the bytecode is used to scale and shift subglyphs. For most fonts, however, this is not the case.

2.3.6 Hint With Components

`--components`, `-c`
: Hint glyph components separately instead of hinting composite glyphs as a whole. Using this flag reduces the bytecode size enormously, however, it might yield worse results. In the GUI it is similar: If you uncheck the 'Process With Components' box, glyph components are hinted separately.

  If a font contains composite glyphs and those glyphs are hinted as a whole, ttfautohint cannot reprocess its own output.

2.3.7 Symbol Font

`--symbol`, `-s`
: Use default values for standard stem width and height instead of deriving them from latin character 'o'. Use this option (usually in combination with option `--latin-fallback`) to hint symbol or dingbat fonts or math glyphs, for example, which lack character 'o', at the expense of possibly poor hinting results at small sizes.

2.3.8 Add ttfautohint Info

`--no-info`, `-n`
: Don't add ttfautohint version and command line information to the version string or strings (with name ID 5) in the font's `name` table. In the GUI it is similar: If you uncheck the 'Add ttfautohint info' box, information is not added to the `name` table. Except for testing and development purposes it is strongly recommended to not use this option.

2.3.9 Strong Stem Width and Positioning

`--strong-stem-width=`*string*, `-w` *string*
: ttfautohint offers two different routines to handle stem widths and stem positions: 'smooth' and

'strong'. The former uses discrete values which slightly increase the stem contrast with almost no distortion of the outlines, while the latter snaps both stem widths and stem positions to integer pixel values as much as possible, yielding a crisper appearance at the cost of much more distortion.

These two routines are mapped onto three possible rendering targets:

- grayscale rendering, with or without optimization for subpixel positioning (e.g. Mac OS X)
- 'GDI ClearType' rendering: the rasterizer version, as returned by the GETINFO bytecode instruction, is in the range 36 <= version < 38 and ClearType is enabled (e.g. Windows XP)
- 'DirectWrite ClearType' rendering: the rasterizer version, as returned by the GETINFO bytecode instruction, is >= 38, ClearType is enabled, and subpixel positioning is enabled also (e.g. Internet Explorer 9 running on Windows 7)

GDI ClearType uses a mode similar to B/W rendering along the vertical axis, while DW ClearType applies grayscale rendering. Additionally, only DW ClearType provides subpixel positioning along the x axis. For what it's worth, the rasterizers version 36 and version 38 in Microsoft Windows are two completely different rendering engines.

The command line option expects *string* to contain up to three letters with possible values 'g' for grayscale, 'G' for GDI ClearType, and 'D' for DW ClearType. If a letter is found in *string*, the strong stem width routine is used for the corresponding rendering target. The default value is 'G' which means that strong stem width handling is activated for GDI ClearType only. To use smooth stem width handling for all three rendering targets, use the empty string as an argument, usually connoted with '""'.

In the GUI, simply set the corresponding check box to select the stem width routine for a given rendering target.

The following FontForge snapshot images use the font 'Mertz Bold' (still under development) from Vernon Adams.

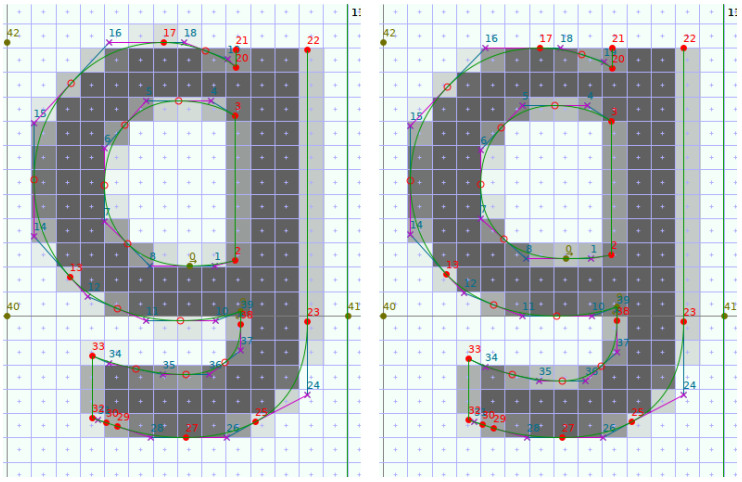

Figure 2.4: The left part shows the glyph 'g' unhinted at 26px, the right part with hints, using the 'smooth' stem algorithm.

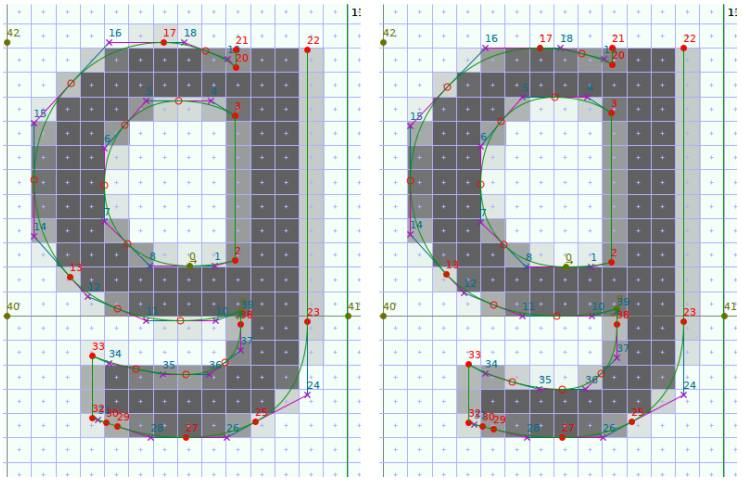

Figure 2.5: The same, but this time using the ‘strong’ algorithm. Note how the stems are aligned to the pixel grid.

2.3.10 Font License Restrictions

`--ignore-restrictions`, `-i`
: By default, fonts which have bit 1 set in the ‘fsType’ field of the `OS/2` table are rejected. If you have a permission of the font’s legal owner to modify the font, specify this command line option.

  If this option is not set, `ttfautohintGUI` shows a dialogue to handle such fonts if necessary.

2.3.11 Miscellaneous

`--help`, `-h`
: On the console, print a brief documentation on standard output and exit. This doesn’t work with `ttfautohintGUI` on MS Windows.

`--version`, `-v`
: On the console, print version information on standard output and exit. This doesn’t work with `ttfautohintGUI` on MS Windows.

`--debug`
: Print *a lot* of debugging information on standard error while processing a font (you should redirect stderr to a file). This doesn’t work with `ttfautohintGUI` on MS Windows.

3 Background and Technical Details

Real-Time Grid Fitting of Typographic Outlines is a scholarly paper which describes FreeType's auto-hinter in some detail. Regarding the described data structures it is slightly out of date, but the algorithm itself hasn't changed.

The next few subsections are mainly based on this article, introducing some important concepts. Note that ttfautohint only does hinting along the vertical direction (this is, modifying y coordinates).

3.1 Segments and Edges

A glyph consists of one or more *contours* (this is, closed curves). For example, glyph 'O' consists of two contours, while glyph 'I' has only one.

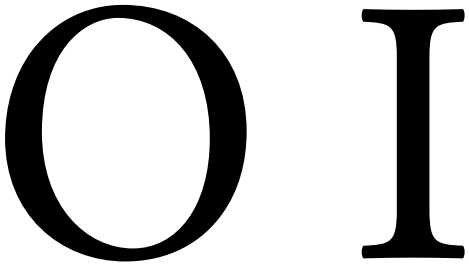

Figure 3.1: The letter 'O' has two contours, an inner and an outer one, while letter 'I' has only an outer contour.

A *segment* is a series of consecutive points of a contour (including its Bézier control points) that are approximately aligned along a coordinate axis.

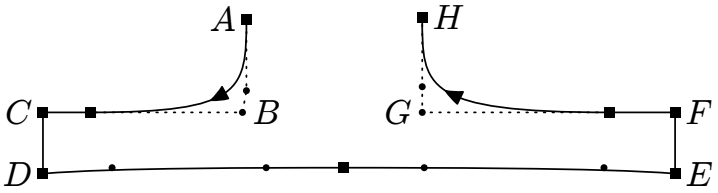

Figure 3.2: A serif. Contour and control points are represented by squares and circles, respectively. The bottom 'line' DE is approximately aligned along the horizontal axis, thus it forms a segment of 7 points. Together with the two other horizontal segments, BC and FG, they form two edges (BC+FG, DE).

An *edge* corresponds to a single coordinate value on the main dimension that collects one or more segments (allowing for a small threshold). While finding segments is done on the unscaled outline, finding edges is bound to the device resolution. See below (section 3.5) for an example.

The analysis to find segments and edges is specific to a script.

3.2 Feature Analysis

The auto-hinter analyzes a font in two steps.

- Global Analysis

  This affects the hinting of all glyphs, trying to give them a uniform appearance.

  – Compute standard stem widths and heights of the font. The values are normally taken from the glyph of letter ‘o’.

  – Compute blue zones, see below (section 3.3).

  If stem widths and heights of single glyphs differ by a large value, or if ttfautohint fails to find proper blue zones, hinting becomes quite poor, leading even to severe shape distortions.

- Glyph Analysis

  This is a per-glyph operation.

  – Find segments and edges.

  – Link edges together to find stems and serifs. The abovementioned paper gives more details on what exactly constitutes a stem or a serif and how the algorithm works.

3.3 Blue Zones

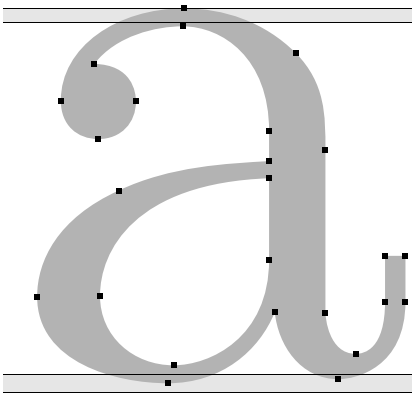

Figure 3.3: Two blue zones relevant to the glyph ‘a’. Vertical point coordinates of *all* glyphs within these zones are aligned.

Outlines of certain characters are used to determine *blue zones*. This concept is the same as with Type 1 fonts: All glyph points which lie in certain small horizontal zones get aligned vertically.

Here a table which shows the characters used by the latin module; the values are hard-coded in the source code.

ID	Blue zone	Characters
1	top of capital letters	THEZOCQS
2	bottom of capital letters	HEZLOCUS
3	top of ‘small f’ like letters	fijkdbh
4	top of small letters	xzroesc
5	bottom of small letters	xzroesc
6	bottom of descenders of small letters	pqgjy

The ‘round’ characters (e.g. ‘OCQS’) from Zones 1, 2, and 5 are also used to control the overshoot handling; to improve rendering at small sizes, zone 4 gets adjusted to be on the pixel grid; cf. the `--increase-x-height` option (subsection 2.3.4).

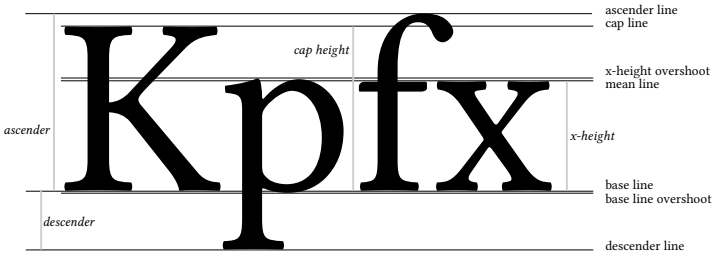

Figure 3.4: This image shows the relevant glyph terms for vertical blue zone positions.

3.4 Grid Fitting

Aligning outlines along the grid lines is called *grid fitting*. It doesn’t necessarily mean that the outlines are positioned *exactly* on the grid, however, especially if you want a smooth appearance at different sizes. This is the central routine of the auto-hinter; its actions are highly dependent on the used script. Currently, only support for scripts which work similarly to Latin (i.e. Greek and Cyrillic) is available.

- Align edges linked to blue zones.
- Fit edges to the pixel grid.
- Align serif edges.
- Handle remaining ‘strong’ points. Such points are not part of an edge but are still important for defining the shape. This roughly corresponds to the `IP` TrueType instruction.
- Everything else (the ‘weak’ points) is handled with an `IUP` instruction.

The following images illustrate the hinting process, using glyph ‘a’ from the freely available font ‘Ubuntu Book’. The manual hints were added by Dalton Maag Ltd, the used application to create the hinting debug snapshots was FontForge.

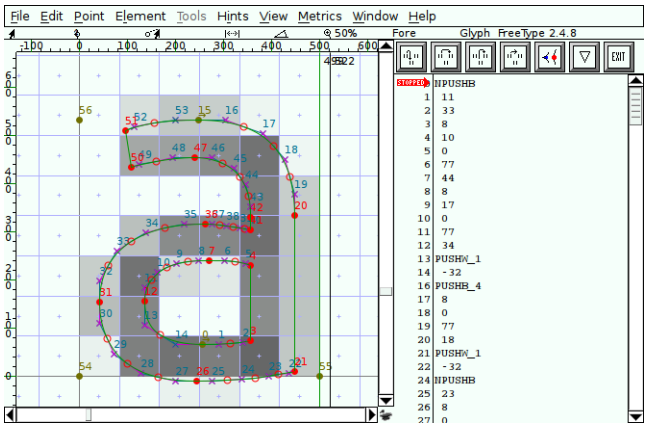

Figure 3.5: Before hinting.

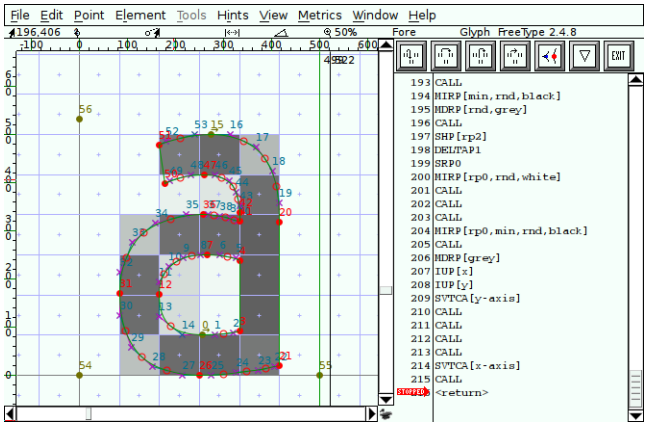

Figure 3.6: After hinting, using manual hints.

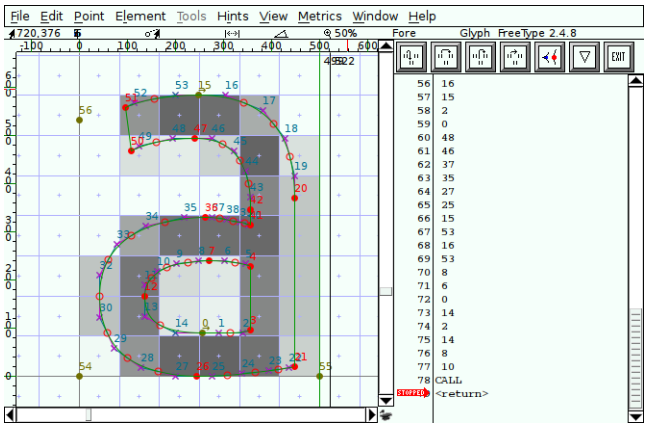

Figure 3.7: After hinting, using ttfautohint. Note that the hinting process doesn't change horizontal positions.

3.5 Hint Sets

In ttfautohint terminology, a *hint set* is the *optimal* configuration for a given PPEM (pixel per EM) value.

In the range given by the `--hinting-range-min` and `--hinting-range-max` options, ttfautohint creates hint sets for every PPEM value. For each glyph, ttfautohint automatically determines if a new set should be emitted for a PPEM value if it finds that it differs from a previous one. For some glyphs it is possible that one set covers, say, the range 8px-1000px, while other glyphs need 10 or more such sets.

In the PPEM range below `--hinting-range-min`, ttfautohint always uses just one set, in the PPEM range between `--hinting-range-max` and `--hinting-limit`, it also uses just one set.

One of the hinting configuration parameters is the decision which segments form an edge. For example, let us assume that two segments get aligned on a single horizontal edge at 11px, while two edges are used at 12px. This change makes ttfautohint emit a new hint set to accomodate this situation.

The next images illustrate this, using a Cyrillic letter (glyph 'afii10108') from the 'Ubuntu book' font, processed with ttfautohint.

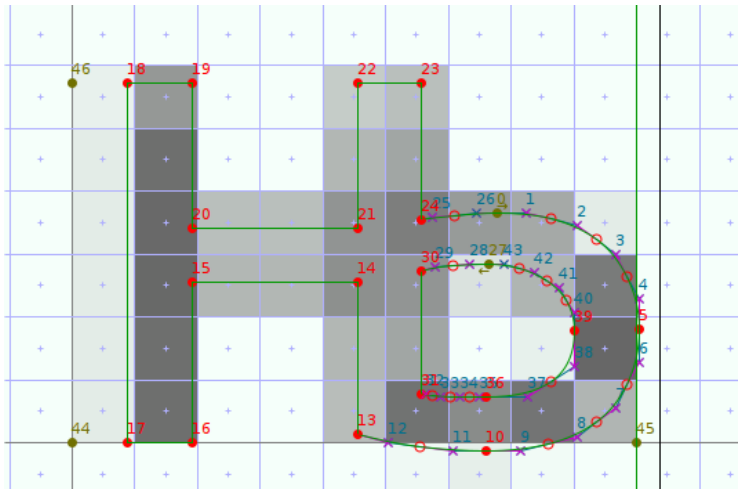

Figure 3.8: Before hinting, size 11px.

Obviously, the more hint sets get emitted, the larger the bytecode ttfautohint adds to the output font. To find a good value *n* for `--hinting-range-max`, some experimentation is necessary since *n* depends on the glyph shapes in the input font. If the value is too low, the hint set created for the PPEM value *n* (this hint set gets used for all larger PPEM values) might distort the outlines too much in the PPEM range given by *n* and the value set by `--hinting-limit` (at which hinting gets switched off). If the value is too high, the font size increases due to more hint sets without any noticeable hinting effects.

Similar arguments hold for `--hinting-range-min` except that there is no lower limit at which hinting is switched off.

An example. Let's assume that we have a hinting range 10 <= ppem <= 100, and the hinting limit is set to 250. For a given glyph, ttfautohint finds out that four hint sets must be computed to exactly cover thes hinting range: 10-15, 16-40, 41-80, and 81-100. For ppem values below 10ppem, the hint set covering 10-15ppem is used, for ppem values larger than 100 the hint set covering 81-100ppem is used. For ppem values larger than 250, no hinting gets applied.

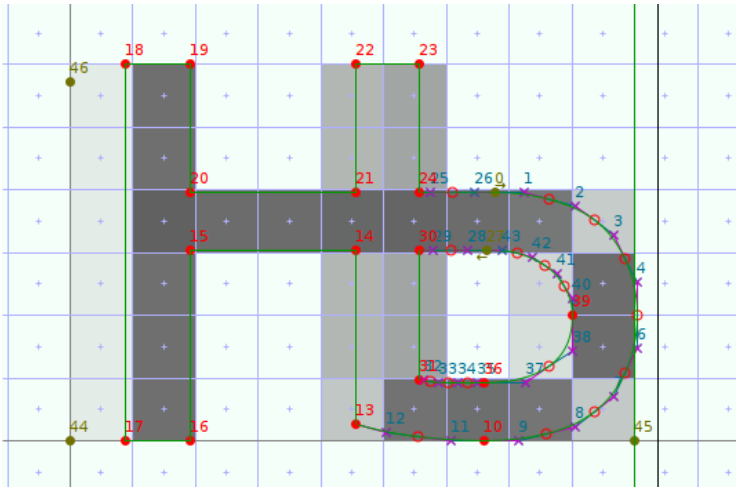

Figure 3.9: After hinting, size 11px. Segments 43-27-28 and 14-15 are aligned on a single edge, as are segments 26-0-1 and 20-21.

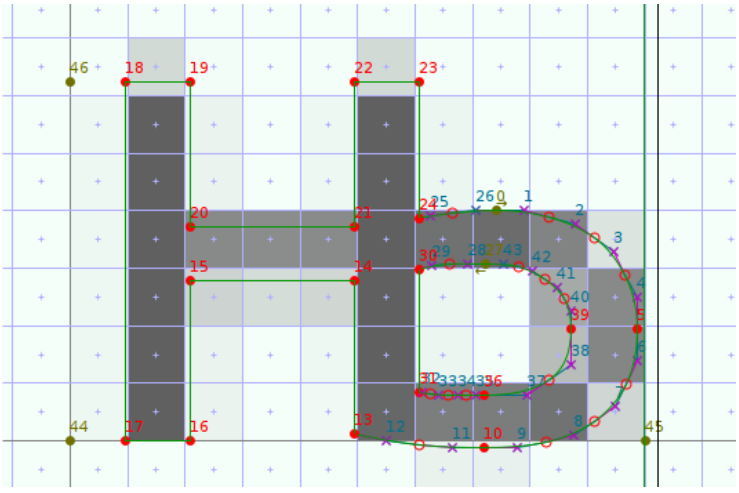

Figure 3.10: Before hinting, size 12px.

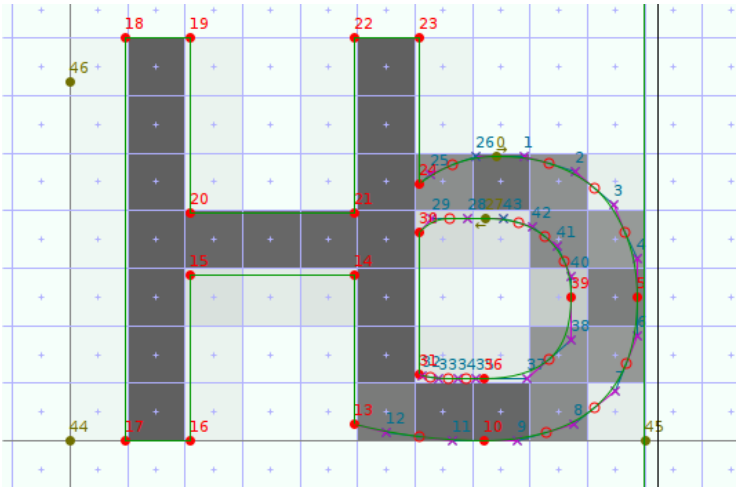

Figure 3.11: After hinting, size 12px. The segments are not aligned. While segments 43-27-28 and 20-21 now have almost the same horizontal position, they don't form an edge because the outlines passing through the segments point into different directions.

3.6 The '.ttfautohint' Glyph

[The behaviour described in this section does not apply if option `--components` (subsection 2.3.6) is used.]

By default, ttfautohint doesn't hint subglyphs of composite glyphs separately. Instead, it hints the whole glyph, this is, composites get recursively expanded internally so that they form simple glyphs, then hints are applied – this is the normal working mode of FreeType's auto-hinter.

One problem, however, must be solved: Hinting for subglyphs (which usually are used as normal glyphs also) must be deactivated so that nothing but the final bytecode of the composite gets executed.

The trick used by ttfautohint is to prepend a composite element called '.ttfautohint', a dummy glyph with a single point, and which has a single job: Its bytecode increases a variable (to be more precise, it is a CVT register called `cvtl_is_subglyph` in the source code), indicating that we are within a composite glyph. The final bytecode of the composite glyph eventually decrements this variable again.

As an example, let's consider composite glyph 'Agrave' ('À') which has the subglyph 'A' as the base and 'grave' as its accent. After processing with ttfautohint it consists of three subglyphs: '.ttfautohint', 'A', and 'grave' (in this order).

Bytecode of	Action
.ttfautohint	increase `cvtl_is_subglyph` (now: 1)
A	do nothing because `cvtl_is_subglyph` > 0
grave	do nothing because `cvtl_is_subglyph` > 0
Agrave	decrease `cvtl_is_subglyph` (now: 0)
	apply hints because `cvtl_is_subglyph` == 0

Some technical details (which you might skip): All glyph point indices get adjusted since each '.ttfautohint' subglyph shifts all following indices by one. This must be done for both the bytecode and one subformat

of OpenType's GPOS anchor tables.

While this approach works fine on all tested platforms, there is one single drawback: Direct rendering of the '.ttfautohint' subglyph (this is, rendering as a stand-alone glyph) disables proper hinting of all glyphs in the font! Under normal circumstances this never happens because '.ttfautohint' doesn't have an entry in the font's cmap table. (However, some test and demo programs like FreeType's ftview application or other glyph viewers which are able to bypass the cmap table might be affected.)

3.7 Scripts

ttfautohint checks which auto-hinting module should be used to hint a specific glyph. To do so, it checks a glyph's Unicode character code whether it belongs to a given script. Currently, only FreeType's 'latin' autohinting module is implemented, but more are expected to come. Here is the hardcoded list of character ranges which are hinted by this 'latin' module. As you can see, this also covers some non-latin scripts (in the Unicode sense) which have similar typographical properties.

Character range	Description
0x0020 - 0x007F	Basic Latin (no control characters)
0x00A0 - 0x00FF	Latin-1 Supplement (no control characters)
0x0100 - 0x017F	Latin Extended-A
0x0180 - 0x024F	Latin Extended-B
0x0250 - 0x02AF	IPA Extensions
0x02B0 - 0x02FF	Spacing Modifier Letters
0x0300 - 0x036F	Combining Diacritical Marks
0x0370 - 0x03FF	Greek and Coptic
0x0400 - 0x04FF	Cyrillic
0x0500 - 0x052F	Cyrillic Supplement
0x1D00 - 0x1D7F	Phonetic Extensions
0x1D80 - 0x1DBF	Phonetic Extensions Supplement
0x1DC0 - 0x1DFF	Combining Diacritical Marks Supplement
0x1E00 - 0x1EFF	Latin Extended Additional
0x1F00 - 0x1FFF	Greek Extended
0x2000 - 0x206F	General Punctuation
0x2070 - 0x209F	Superscripts and Subscripts
0x20A0 - 0x20CF	Currency Symbols
0x2150 - 0x218F	Number Forms
0x2460 - 0x24FF	Enclosed Alphanumerics
0x2C60 - 0x2C7F	Latin Extended-C
0x2DE0 - 0x2DFF	Cyrillic Extended-A
0x2E00 - 0x2E7F	Supplemental Punctuation
0xA640 - 0xA69F	Cyrillic Extended-B
0xA720 - 0xA7FF	Latin Extended-D
0xFB00 - 0xFB06	Alphabetical Presentation Forms (Latin Ligatures)
0x1D400 - 0x1D7FF	Mathematical Alphanumeric Symbols
0x1F100 - 0x1F1FF	Enclosed Alphanumeric Supplement

If a glyph's character code is not covered by a script range, it is not hinted (or rather, it gets hinted by the 'dummy' auto-hinting module which essentially does nothing). This can be changed by specifying a *fallback script* with option `--latin-fallback` (subsection 2.3.2).

It is planned to extend ttfautohint so that the `GSUB` OpenType table gets analyzed, mapping character codes to all glyph indices which can be reached by switching on or off various OpenType features.

3.8 SFNT Tables

ttfautohint touches almost all SFNT tables within a TrueType or OpenType font. Note that only OpenType fonts with TrueType outlines are supported. OpenType fonts with a `CFF` table (this is, with PostScript outlines) won't work.

- `glyf`: One glyph gets added (namely the '.ttfautohint' glyph); all composites get an additional component; all hints in the table are replaced with new ones.
- `cvt`, `prep`, and `fpgm`: These tables get replaced with data necessary for the new hinting bytecode.
- `gasp`: Set up to always use grayscale rendering with grid-fitting for standard hinting, and symmetric grid-fitting and symmetric smoothing for horizontal subpixel hinting (ClearType).
- `DSIG`: If it exists, it gets replaced with a dummy version. ttfautohint can't digitally sign a font; you have to do that afterwards.
- `name`: The 'version' entries are modified to add information about the parameters which have been used for calling ttfautohint. This can be controlled with the `--no-info` (??) option.
- `GPOS`, `hmtx`, `loca`, `head`, `maxp`, `post`: Updated to fit the additional '.ttfautohint' glyph, the additional subglyphs in composites, and the new hinting bytecode.
- `LTSH`, `hdmx`: Since ttfautohint doesn't do any horizontal hinting, those tables are superfluous and thus removed.
- `VDMX`: Removed, since it depends on the original bytecode which ttfautohint removes. A font editor might recompute the necessary data later on.

3.9 Problems

Diagonals.

TODO

4 The ttfautohint API

This section documents the single function of the ttfautohint library, `TTF_autohint`, together with its callback functions, `TA_Progress_Func` and `TA_Info_Func`. All information has been directly extracted from the `ttfautohint.h` header file.

4.1 Preprocessor Macros and Typedefs

Some default values.

```
#define TA_HINTING_RANGE_MIN 8
#define TA_HINTING_RANGE_MAX 50
#define TA_HINTING_LIMIT 200
#define TA_INCREASE_X_HEIGHT 14
```

An error type.

```
typedef int TA_Error;
```

4.2 Callback: `TA_Progress_Func`

A callback function to get progress information. *curr_idx* gives the currently processed glyph index; if it is negative, an error has occurred. *num_glyphs* holds the total number of glyphs in the font (this value can't be larger than 65535).

curr_sfnt gives the current subfont within a TrueType Collection (TTC), and *num_sfnts* the total number of subfonts.

If the return value is non-zero, `TTF_autohint` aborts with `TA_Err_Canceled`. Use this for a 'Cancel' button or similar features in interactive use.

progress_data is a void pointer to user supplied data.

```
typedef int
(*TA_Progress_Func)(long curr_idx,
                    long num_glyphs,
                    long curr_sfnt,
                    long num_sfnts,
                    void* progress_data);
```

4.3 Callback: TA_Info_Func

A callback function to manipulate strings in the name table. *platform_id*, *encoding_id*, *language_id*, and *name_id* are the identifiers of a name table entry pointed to by *str* with a length pointed to by *str_len* (in bytes; the string has no trailing NULL byte). Please refer to the OpenType specification for a detailed description of the various parameters, in particular which encoding is used for a given platform and encoding ID.

The string *str* is allocated with malloc; the application should reallocate the data if necessary, ensuring that the string length doesn't exceed 0xFFFF.

info_data is a void pointer to user supplied data.

If an error occurs, return a non-zero value and don't modify *str* and *str_len* (such errors are handled as non-fatal).

```
typedef int
(*TA_Info_Func)(unsigned short platform_id,
                unsigned short encoding_id,
                unsigned short language_id,
                unsigned short name_id,
                unsigned short* str_len,
                unsigned char** str,
                void* info_data);
```

4.4 Function: TTF_autohint

Read a TrueType font, remove existing bytecode (in the SFNT tables prep, fpgm, cvt, and glyf), and write a new TrueType font with new bytecode based on the autohinting of the FreeType library.

It expects a format string *options* and a variable number of arguments, depending on the fields in *options*. The fields are comma separated; whitespace within the format string is not significant, a trailing comma is ignored. Fields are parsed from left to right; if a field occurs multiple times, the last field's argument wins. The same is true for fields which are mutually exclusive. Depending on the field, zero or one argument is expected.

Note that fields marked as 'not implemented yet' are subject to change.

`in-file`
: A pointer of type FILE* to the data stream of the input font, opened for binary reading. Mutually exclusive with in-buffer.

`in-buffer`
: A pointer of type const char* to a buffer which contains the input font. Needs in-buffer-len. Mutually exclusive with in-file.

`in-buffer-len`
: A value of type size_t, giving the length of the input buffer. Needs in-buffer.

`out-file`
: A pointer of type `FILE*` to the data stream of the output font, opened for binary writing. Mutually exclusive with `out-buffer`.

`out-buffer`
: A pointer of type `char**` to a buffer which contains the output font. Needs `out-buffer-len`. Mutually exclusive with `out-file`. Deallocate the memory with `free`.

`out-buffer-len`
: A pointer of type `size_t*` to a value giving the length of the output buffer. Needs `out-buffer`.

`progress-callback`
: A pointer of type `TA_Progress_Func` (section 4.2), specifying a callback function for progress reports. This function gets called after a single glyph has been processed. If this field is not set or set to NULL, no progress callback function is used.

`progress-callback-data`
: A pointer of type `void*` to user data which is passed to the progress callback function.

`error-string`
: A pointer of type `unsigned char**` to a string (in UTF-8 encoding) which verbally describes the error code. You must not change the returned value.

`hinting-range-min`
: An integer (which must be larger than or equal to 2) giving the lowest PPEM value used for autohinting. If this field is not set, it defaults to `TA_HINTING_RANGE_MIN`.

`hinting-range-max`
: An integer (which must be larger than or equal to the value of `hinting-range-min`) giving the highest PPEM value used for autohinting. If this field is not set, it defaults to `TA_HINTING_RANGE_MAX`.

`hinting-limit`
: An integer (which must be larger than or equal to the value of `hinting-range-max`) which gives the largest PPEM value at which hinting is applied. For larger values, hinting is switched off. If this field is not set, it defaults to `TA_HINTING_LIMIT`. If it is set to 0, no hinting limit is added to the bytecode.

`gray-strong-stem-width`
: An integer (1 for 'on' and 0 for 'off', which is the default) which specifies whether horizontal stems should be snapped and positioned to integer pixel values for normal grayscale rendering.

`gdi-cleartype-strong-stem-width`
: An integer (1 for 'on', which is the default, and 0 for 'off') which specifies whether horizontal stems should be snapped and positioned to integer pixel values for GDI ClearType rendering, this is, the rasterizer version (as returned by the GETINFO bytecode instruction) is in the range 36 <= version < 38 and ClearType is enabled.

`dw-cleartype-strong-stem-width`
: An integer (1 for 'on' and 0 for 'off', which is the default) which specifies whether horizontal stems should be snapped and positioned to integer pixel values for DW ClearType rendering, this is, the rasterizer version (as returned by the GETINFO bytecode instruction) is >= 38, ClearType is enabled, and subpixel positioning is enabled also.

`increase-x-height`
: An integer. For PPEM values in the range 6 <= PPEM <= `increase-x-height`, round up the font's x height much more often than normally. If it is set to 0, this feature is switched off. If this field is not set, it defaults to `TA_INCREASE_X_HEIGHT`. Use this flag to improve the legibility of small font sizes if necessary.

`hint-with-components`
: If this integer is set to 1 (which is the default), ttfautohint handles composite glyphs as a whole. This implies adding a special glyph to the font, as documented here (section 3.6). Setting it to 0, the components of composite glyphs are hinted separately. While separate hinting of subglyphs makes the resulting bytecode much smaller, it might deliver worse results. However, this depends on the processed font and must be checked by inspection.

`pre-hinting`
: An integer (1 for 'on' and 0 for 'off', which is the default) to specify whether native TrueType hinting shall be applied to all glyphs before passing them to the (internal) autohinter. The used resolution is the em-size in font units; for most fonts this is 2048ppem. Use this if the hints move or scale subglyphs independently of the output resolution.

`info-callback`
: A pointer of type `TA_Info_Func` (section 4.3), specifying a callback function for manipulating the `name` table. This function gets called for each `name` table entry. If not set or set to NULL, the table data stays unmodified.

`info-callback-data`
: A pointer of type `void*` to user data which is passed to the info callback function.

`x-height-snapping-exceptions`
: A pointer of type `const char*` to a null-terminated string which gives a list of comma separated PPEM values or value ranges at which no x-height snapping shall be applied. A value range has the form *value1*-*value2*, meaning *value1* <= PPEM <= *value2*. Whitespace is not significant; a trailing comma is ignored. If the supplied argument is NULL, no x-height snapping takes place at all. By default, there are no snapping exceptions. Not implemented yet.

`ignore-restrictions`
: If the font has set bit 1 in the 'fsType' field of the `OS/2` table, the ttfautohint library refuses to process the font since a permission to do that is required from the font's legal owner. In case you have such a permission you might set the integer argument to value 1 to make ttfautohint handle the font. The default value is 0.

`fallback-script`
: An integer which specifies the default script for glyphs not in the 'latin' range. If set to 1, the 'latin' script is used (other scripts are not supported yet). By default, no script is used (value 0; this disables autohinting for such glyphs).

`symbol`
: Set this integer to 1 if you want to process a font which lacks the characters of a supported script, for example, a symbol font. ttfautohint then uses default values for the standard stem width and height instead of deriving these values from a script's key character (for the latin script, it is character 'o'). The default value is 0.

`debug`
: If this integer is set to 1, lots of debugging information is print to stderr. The default value is 0.

Remarks:

- Obviously, it is necessary to have an input and an output data stream. All other options are optional.
- `hinting-range-min` and `hinting-range-max` specify the range for which the autohinter generates optimized hinting code. If a PPEM value is smaller than the value of `hinting-range-min`, hinting still takes place but the configuration created for `hinting-range-min` is used. The analogous action is taken for `hinting-range-max`, only limited by the value given with `hinting-limit`. The font's `gasp` table is set up to always use grayscale rendering with grid-fitting for standard hinting, and symmetric grid-fitting and symmetric smoothing for horizontal subpixel hinting (ClearType).
- ttfautohint can't process a font a second time (well, it can, if the font doesn't contain composite glyphs). Just think of ttfautohint as being a compiler, a tool which also can't process its created output again.

```
TA_Error
TTF_autohint(const char* options,
             ...);
```

5 Compilation and Installation

Please read the files `INSTALL` and `INSTALL.git` (part of the source code bundle) for instructions how to compile the ttfautohint library together with its front-ends.

TODO

5.1 Unix Platforms

TODO

5.2 MS Windows

TODO

5.3 Mac OS X

TODO

6 Authors