

ttfautohint

Werner Lemberg

Version 1.2

Contents

1	Introduction	5
1.1	What exactly are hints?	5
1.2	What problems can arise with TrueType hinting?	6
1.3	Why ttfautohint?	6
2	ttfautohint and ttfautohintGUI	8
2.1	Calling ttfautohint	8
2.2	Calling ttfautohintGUI	8
2.3	Options	8
2.3.1	Hint Set Range Minimum, Hint Set Range Maximum	10
2.3.2	Default Script	10
2.3.3	Fallback Script	10
2.3.4	Hinting Limit	10
2.3.5	x Height Increase Limit	11
2.3.6	x Height Snapping Exceptions	12
2.3.7	Fallback Stem Width	12
2.3.8	Windows Compatibility	12
2.3.9	Adjust Subglyphs	13
2.3.10	Hint Composites	13
2.3.11	Symbol Font	13
2.3.12	Dehint	13
2.3.13	Add ttfautohint Info	14
2.3.14	Add TTFA Info Table	14
2.3.15	Strong Stem Width and Positioning	14
2.3.16	Control Instructions File	15
2.3.17	Miscellaneous	19
3	Background and Technical Details	20
3.1	Segments and Edges	20
3.2	Feature Analysis	21
3.3	Blue Zones	21
3.4	Grid Fitting	22
3.5	Hint Sets	23
3.6	Composite Glyphs	27
3.7	The ‘ttfautohint’ Glyph	27
3.8	Writing Systems	28
3.9	Scripts	28
3.10	OpenType Features	30
3.11	SFNT Tables	31

3.12	Problems	31
3.12.1	Interaction With FreeType	31
3.12.2	Incorrect Unicode Character Map	32
3.12.3	Irregular Glyph Heights	32
3.12.4	Diagonals	32
4	The ttfautohint API	33
4.1	Preprocessor Macros and Typedefs	33
4.2	Callback: TA_Progress_Func	33
4.3	Callback: TA_Error_Func	34
4.4	Callback: TA_Info_Func	34
4.5	Function: TTF_autohint	35
4.5.1	I/O	35
4.5.2	Messages and Callbacks	36
4.5.3	General Hinting Options	37
4.5.4	Hinting Algorithms	38
4.5.5	Scripts	39
4.5.6	Miscellaneous	39
4.5.7	Remarks	40
5	Compilation and Installation	41
5.1	Unix-like Platforms	41
5.2	MS Windows	41
5.3	Mac OS X	41
6	Authors	42
7	History	43
7.1	New in Version 1.2 (2014-Oct-06)	43
7.2	New in Version 1.1 (2014-May-05)	43
7.3	New in Version 1.00 (2014-Mar-20)	44
7.4	New in Version 1.00rc1 (2014-Feb-07)	44
7.5	New in Version 0.97 (2013-Nov-09)	44
7.6	New in Version 0.96 (2013-Aug-06)	45
7.7	New in Version 0.95 (2013-Mar-07)	45
7.8	New in Version 0.94 (2012-Nov-29)	45
7.9	New in Version 0.93 (2012-Oct-09)	45
7.10	New in Version 0.92 (2012-Aug-07)	46
7.11	New in Version 0.91 (2012-Jul-12)	46
7.12	New in Version 0.9 (2012-Jun-06)	46
7.13	New in Version 0.8 (2012-Mar-21)	46
7.14	New in Version 0.7 (2012-Feb-05)	47
7.15	New in Version 0.6.1 (2012-Jan-02)	47
7.16	New in Version 0.6 (2011-Dec-25)	47
7.17	New in Version 0.5 (2011-Nov-06)	47
7.18	New in Version 0.4 (2011-Oct-27)	47
7.19	New in Version 0.3 (2011-Sep-09)	47
7.20	New in Version 0.2 (2011-Jul-19)	47

7.21 New in Version 0.1 (2011-Jun-30)	48
---	----

1 Introduction

ttfautohint is a library written in C that 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 auto-hinting module. The idea is to provide the excellent quality of the auto-hinter on platforms that 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 program 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 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 that 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 that 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 (including Android, ChromeOS, and other mobile operating systems) that use FreeType for rendering glyphs.

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.

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` ([subsection 2.3.10](#)) is 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 latn 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 latn > 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`.

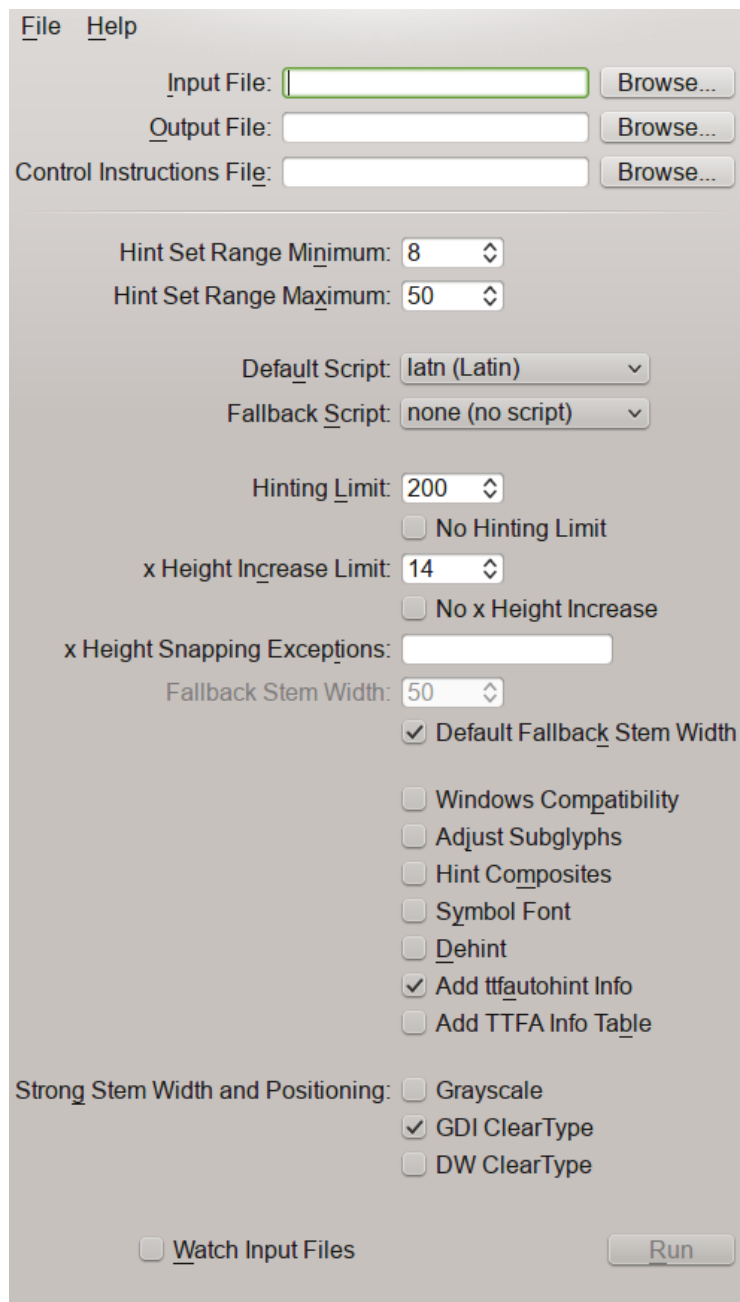


Figure 2.1: ttfautohintGUI on GNU/Linux running KDE

Below, the section title refers to the command's label in the GUI (if applicable), 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.

Increasing the range given by `-l` and `-r` normally makes the font's bytecode larger.

2.3.2 Default Script

`--default-script=s, -D s`

Set default script to tag *s*, which is a string consisting of four lowercase characters like `latn` or `df1t`. It is needed to specify the OpenType default script: After applying all features that are handled specially (like small caps or superscript), `ttfautohint` uses this value for the remaining features. The default value is `latn`. See below ([section 3.10](#)) for more details.

2.3.3 Fallback Script

`--fallback-script=s, -f s`

Set fallback script to tag *s*, which is a string consisting of four characters like `latn` or `df1t`. It gets used for all glyphs that can't be assigned to a script automatically. See below ([section 3.9](#)) for more details.

2.3.4 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, not the `gasp` table data); it does not influence 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 causes internal math overflow in the rasterizer for large pixel values (> 1500px approx.) it is strongly recommended to not use this except for testing purposes.

2.3.5 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 (to use the terminology of TrueType's 'Super Round' bytecode instruction, the threshold is $5/8$ px). If this flag is set, values in the range 6 PPEM to *n* PPEM are much more often rounded up (setting the threshold to $13/16$ px). 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). To switch off rounding the x height to the pixel grid in general, either partially or completely, see 'x Height Snapping Exceptions' ([subsection 2.3.6](#)).

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

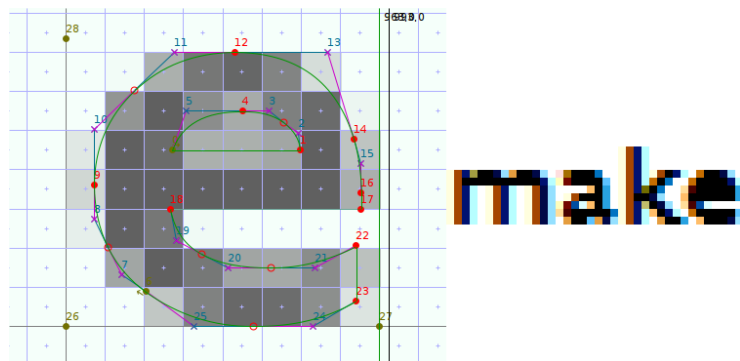


Figure 2.2: At 17px, without option `-x` and `'-w "'`, the hole in glyph 'e' looks very grey in the FontForge snapshot, and the GDI ClearType rendering (which is the default on older Windows versions) fills it completely with black because it uses B/W rendering along the y axis. FreeType's 'light' autohint 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.

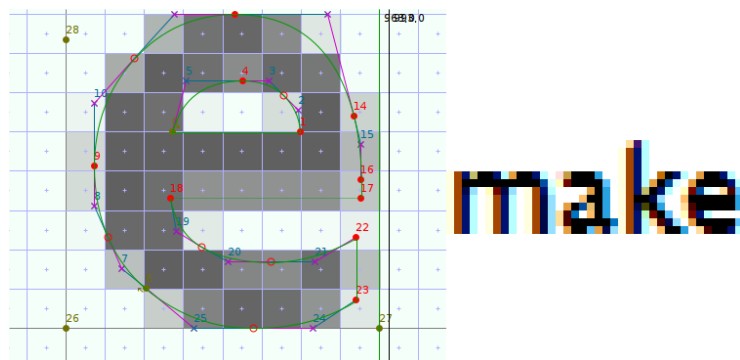


Figure 2.3: The same, this time with option `-x 17` (and `'-w "'`).

2.3.6 x Height Snapping Exceptions

`--x-height-snapping-exceptions=string, -X string`

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*. *value₁* or *value₂* (or both) can be missing; a missing value is replaced by the beginning or end of the whole interval of valid PPEM values, respectively (6 to 32767). Whitespace is not significant; superfluous commas are ignored, and ranges must be specified in increasing order. For example, the string "7-9, 11, 13-" means the values 7, 8, 9, 11, 13, 14, 15, etc. Consequently, if the supplied argument is "-", no x height snapping takes place at all. The default is the empty string (""), meaning no snapping exceptions.

Normally, x height snapping means a slight increase in the overall vertical glyph size so that the height of lowercase glyphs gets aligned to the pixel grid (this is a global feature, affecting *all* glyphs of a font). However, having larger vertical glyph sizes is not always desired, especially if it is not possible to adjust the `usWinAscent` and `usWinDescent` values from the font's OS/2 table so that they are not too tight. See 'Windows Compatibility ([subsection 2.3.8](#))' for more details.

2.3.7 Fallback Stem Width

`--fallback-stem-width=n, -H n`

Set the horizontal stem width (hinting) value for all scripts that lack proper standard characters in the font. The value is given in font units and must be a positive integer. If not set, `ttfautohint` uses a hard-coded default (50 units at 2048 units per EM, and linearly scaled for other UPEM values, for example 24 units at 1000 UPEM).

For symbol fonts, you need option `--fallback-script` too (to set up a script at all).

In the GUI, uncheck the 'Default Fallback Stem Width' box to activate this feature.

2.3.8 Windows Compatibility

`--windows-compatibility, -W`

This option makes `ttfautohint` add two artificial blue zones, positioned at the `usWinAscent` and `usWinDescent` values (from the font's OS/2 table). The idea is to help `ttfautohint` so that the hinted glyphs stay within this horizontal stripe since Windows clips everything falling outside.

There is a general problem with tight values for `usWinAscent` and `usWinDescent`; a good description is given in the [Vertical Metrics How-To](#). Additionally, there is a special problem with tight values if used in combination with `ttfautohint` because the auto-hinter tends to slightly increase the vertical glyph dimensions at smaller sizes to improve legibility. This enlargement can make the heights and depths of glyphs exceed the range given by `usWinAscent` and `usWinDescent`.

If `ttfautohint` is part of the font creation tool chain, and the font designer can adjust those two values, a better solution instead of using option `-W` is to reserve some vertical space for 'padding': For the auto-hinter, the difference between a top or bottom outline point before and after hinting is less than 1px, thus a vertical padding of 2px is sufficient. Assuming a minimum hinting size of 6ppem, adding two pixels gives an increase factor of $8 \div 6 = 1.33$. This is near to the default baseline-to-baseline distance used by TeX and other sophisticated text processing applications, namely $1.2 \times \text{designsize}$, which gives satisfying results in most cases. It is also near to the factor

1.25 recommended in the abovementioned how-to. For example, if the vertical extension of the largest glyph is 2000 units (assuming that it approximately represents the designsize), the sum of `usWinAscent` and `usWinDescent` could be $1.25 \times 2000 = 2500$.

In case `ttfautohint` is used as an auto-hinting tool for fonts that can be no longer modified to change the metrics, option `-W` in combination with `'-X "-"` to suppress any vertical enlargement should prevent almost all clipping.

2.3.9 Adjust Subglyphs

`--adjust-subglyphs, -p`

Adjusting subglyphs makes a font's original bytecode be 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 that modify the shape even at EM size (normally 2048px); in particular, some CJK fonts need this because the bytecode is used to scale and shift subglyphs (hence the option's long name). For most fonts, however, this is not the case.

2.3.10 Hint Composites

`--composites, -c`

By default, the components of a composite glyph get hinted separately. If this flag is set, the composite glyph itself gets hinted (and the hints of the components are ignored). Using this flag increases the bytecode size a lot, however, it might yield better hinting results.

If this option is used (and a font actually contains composite glyphs), `ttfautohint` currently cannot reprocess its own output for technical reasons, see below ([section 3.7](#)).

2.3.11 Symbol Font

`--symbol, -s`

Process a font that `ttfautohint` would refuse otherwise because it can't find a single standard character for any of the supported scripts.

For all scripts that lack proper standard characters, `ttfautohint` uses a default (hinting) value for the standard stem width instead of deriving it from a script's set of standard characters (for the latin script, one of them is character 'o').

Use this option (usually in combination with option `--fallback-script`) to hint symbol or dingbat fonts or math glyphs, for example, at the expense of possibly poor hinting results at small sizes.

2.3.12 Dehint

`--dehint, -d`

Strip off all hints without generating new hints. Consequently, all other hinting options are ignored. This option is intended for testing purposes.

2.3.13 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.

2.3.14 Add TTFA Info Table

`--ttfa-info, -t`

Add an SFNT table called TTFA to the output font that holds a dump of all parameters; the data resembles the format of the `--debug` option's parameter listing. In particular, it lists all ttfautohint control instructions (which are *not* shown in the name table info). This option is mainly for archival purposes so that all information used to create a font is stored in the font itself. Note that such a TTFA table gets ignored by all TrueType rendering engines.

Forthcoming versions of the ttfautohint front-ends will be able to use this data so that a font can be processed another time with exactly the same parameters, thus providing a means for round-tripping fonts.

2.3.15 Strong Stem Width and Positioning

`--strong-stem-width=string, -w string`

ttfautohint offers two different routines to handle (horizontal) stem widths and stem positions: 'smooth' and 'strong'. The former uses discrete values that 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. Android)
- 'GDI ClearType' rendering: the rasterizer version, as returned by the GETINFO bytecode instruction, is in the range $36 \leq \text{version} < 38$ 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 (and smooth stem width handling otherwise). 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 strong width routine for a given rendering target. If you unset the check box, the smooth width routine gets used.

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

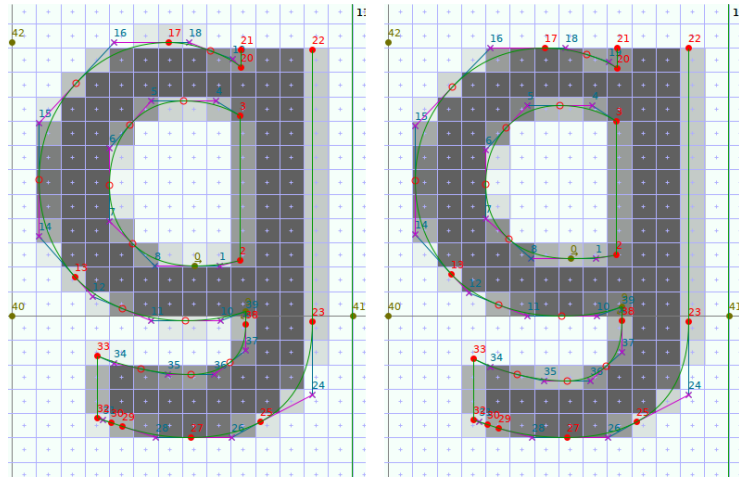


Figure 2.4: The left part shows the glyph 'g' unhinted at 26px, the right part with hints, using the 'smooth' stem algorithm.

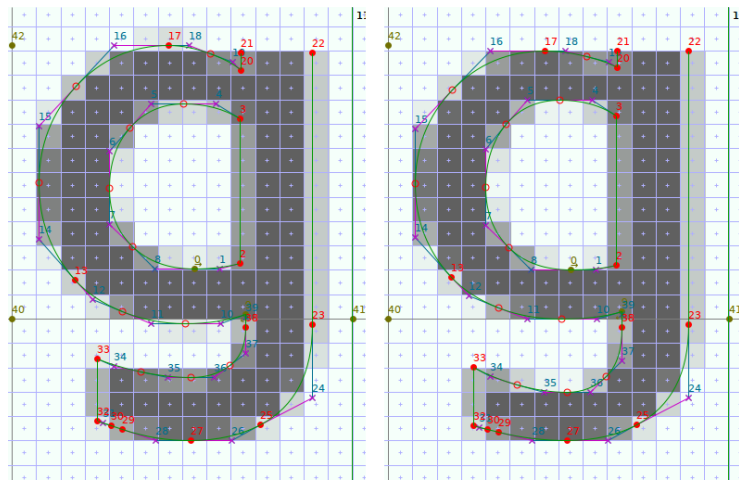


Figure 2.5: The same, but this time using the 'strong' algorithm. Note how the stems are aligned to the pixel grid.

2.3.16 Control Instructions File

`--control-file=file, -m file` (TTY only)

Specify the name of a control instructions file to manually tweak the hinting process. This feature can be used to correct glitches in ttfautohint's hinting algorithm.

An entry in a control instructions file has one of the following syntax forms:

```
[ font-idx ] glyph-id l|r points [ ( left-offset , right-offset ) ]
[ font-idx ] glyph-id n points
[ font-idx ] glyph-id p points [ x x-shift ] [ y y-shift ] @ ppems
```

font-idx gives the index of the font in a TrueType Collection. If missing, it is set to zero. For normal TrueType fonts, only value zero is valid. If starting with 0x the number is interpreted as hexadecimal. If starting with 0 it gets interpreted as an octal value, and as a decimal value otherwise.

glyph-id is a glyph's name as listed in the post SFNT table or a glyph index. A glyph name consists of characters from the set 'A-Za-z0-9._' only and does not start with a digit or period, with the exceptions of the names '.notdef' and '.null'. A glyph index can be specified in decimal, octal, or hexadecimal format, the latter two indicated by the prefixes 0 and 0x, respectively.

The mutually exclusive parameters 'l' and 'r' indicate that the following points have left or right 'out' direction, respectively, overriding ttfautohint's algorithm for setting point directions. The 'out direction' of a point is the direction of the outline *leaving* the point (or passing the control point). If the specified direction is identical to what ttfautohint computes, nothing special happens. Otherwise, a one-point segment with the specified direction gets created. By default, its length is zero. Setting *left-offset* and *right-offset*, you can change the segment's horizontal start and end position relative to the point position. *left-offset* and *right-offset* are integers measured in font units.

Parameter 'n' sets the 'out' direction of the following points to 'no direction'. If the specified direction is identical to what ttfautohint computes, nothing special happens. Otherwise, ttfautohint no longer considers those points as part of horizontal segments, thus treating them as 'weak' points.

Modifying or adding segments don't directly modify the outlines; it only influences the hinting process.

Parameter 'p' makes ttfautohint apply delta exceptions for the given points, shifting the points by the given values. Note that those delta exceptions are applied *after* the final IUP instructions in the bytecode; as a consequence, they are (partially) ignored by rasterizers if in ClearType mode.

Both *points* and *ppems* are number ranges, see 'x Height Snapping Exceptions ([subsection 2.3.6](#))' for the syntax.

x-shift and *y-shift* represent real numbers that get rounded to multiples of 1/8 pixels. The entries for 'x' and 'y' are optional; if missing, the corresponding value is set to zero.

Values for *x-shift* and *y-shift* must be in the range [-1.0;1.0]. Values for *ppems* must be in the range [6;53]. Values for *points* are limited by the number of points in the glyph.

Similar to the Bourne shell (sh or bash), a comment starts with character '#'; the rest of the line is ignored. An empty line is ignored also. Both the newline character and ';' can be used as a separator between exception entries. A trailing '\' at the end of a line continues the current line on the next one.

In case there are multiple shift entries for the same (*font idx*, *glyph-id*, *point-idx*, *ppem-value*) quadruplet, the first entry in the control instructions file wins.

Note that only character '.' is recognized as a decimal point, and a thousands separator is not accepted.

As an example for delta instructions, let's assume that you want to shift points 2, 3, and 4 in glyph 'Aacute' at ppe sizes 12 and 13 by a vertical amount of 0.25 pixels. This corresponds to the line

```
Aacute p 2-4 y 0.25 @ 12, 13
```

in a control instructions file.

The following images display glyphs from the font **Halant-Regular**.

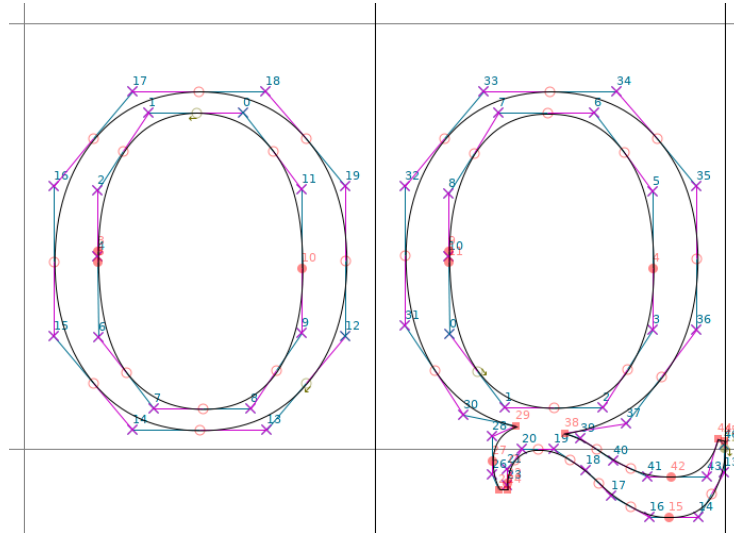


Figure 2.6: The outlines of glyphs 'O' and 'Q', as displayed in FontForge. They are sufficiently similar to expect that ttfautohint hints them equally. However, this is not the case.

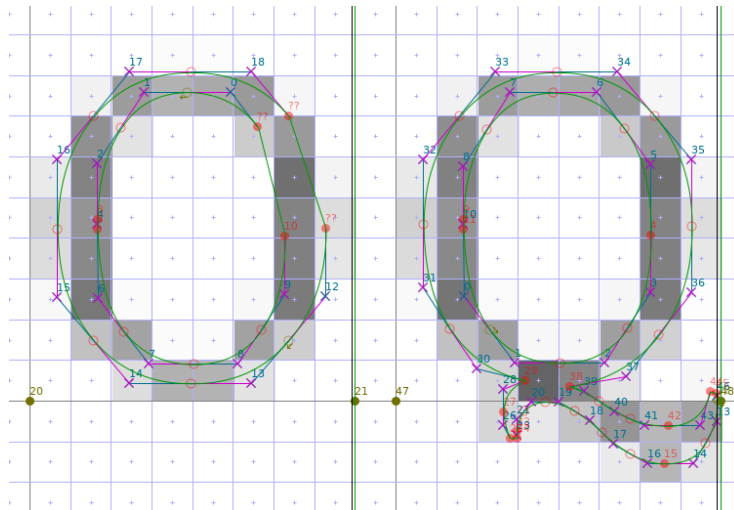


Figure 2.7: The same glyphs, shown at 12px before hinting – please ignore the outline distortion in the upper right of glyph 'O'; this is a bug in FontForge while running the TrueType debugger.

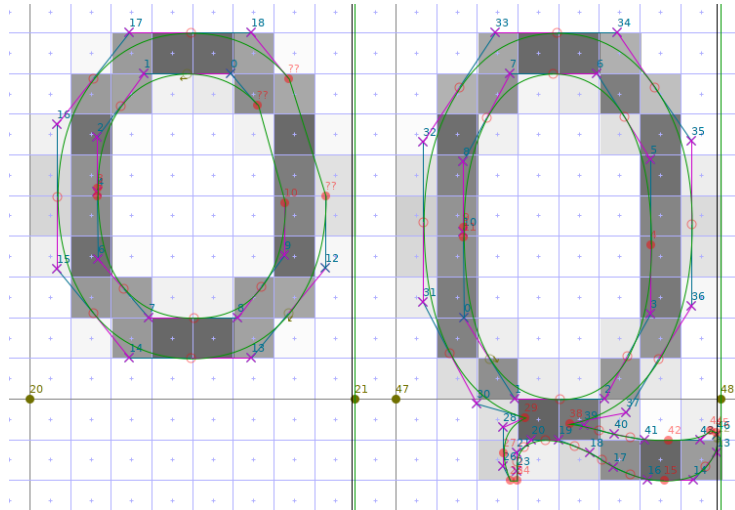


Figure 2.8: Using only ttfautohint's `-w gGD` parameter to force strong stem width and positioning, the hinting of glyph 'Q' is really bad, making the glyph vertically two pixels larger! Reason is that this glyph doesn't contain a horizontal segment at the baseline blue zone ($y = 1$; this corresponds to the segment 13-14 in the 'O' glyph). Normally, segment 1-2 would form a 'stem' with the baseline segment (as segment 7-8 does in glyph 'O'). Instead, it forms a stem with segment 19-20, which gets moved down ($y = -1$) because the whole glyph appears to be stretched.

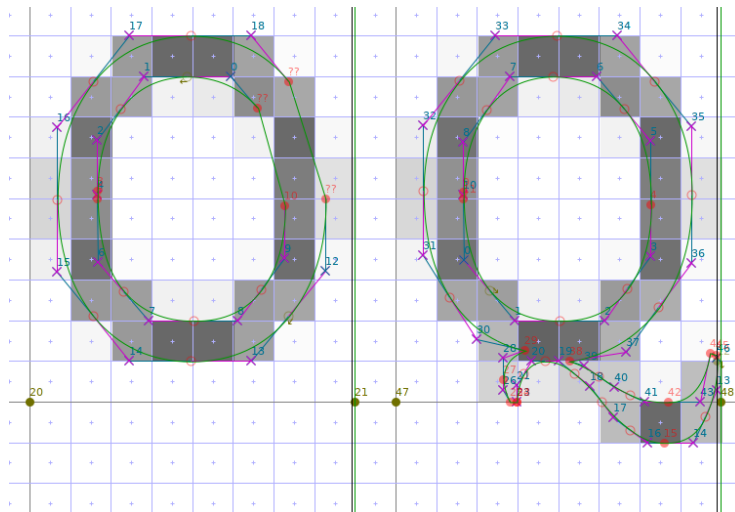


Figure 2.9: To fix the problem, we change the direction of point 38 to 'left' by writing a line `'Q 1 38'` (without the quotes) to a control description file `Halant-Regular.txt`. Adding option `'-m Halant-Regular.txt'` to ttfautohint, we get the shown image as a result, which is much better: Segment 1-2 now properly forms a stem with our artificial one-point segment 38, and the 'O'-like shape is properly positioned. However, there is still room for improvement: Segment 19-20 is also positioned at the baseline, making the connection between the 'O' shape and the tail too thin.

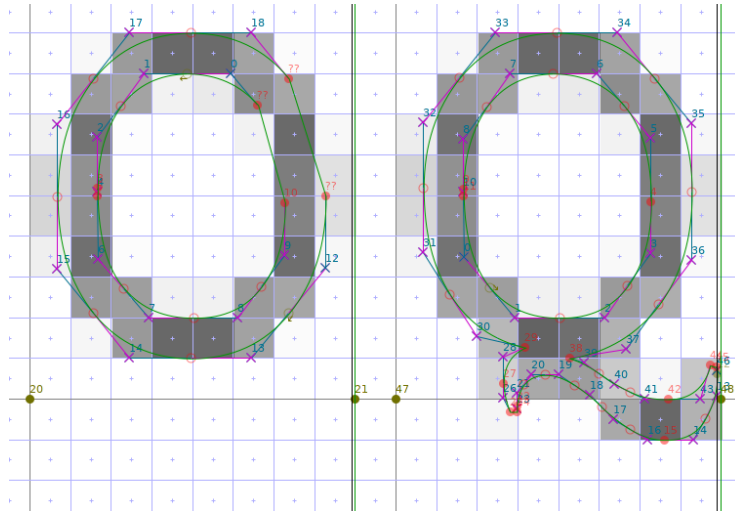


Figure 2.10: By giving the one-point segment 38 a horizontal width, we can prevent that segment 19-20 gets positioned at the baseline: Replace the line in the previous image description with ‘Q 1 38 (-70,20)’, making the segment extend 70 font units to the left and 20 to the right of point 38. The exact offset values don’t matter; it’s only important to start left of point 19. Another solution to the problem is to artificially change the direction of segment 19-20 by adding a second line ‘Q r 19-20’ to the control instructions file; for our ‘Q’ glyph, this produces almost exactly the same hinting results. Note that such direction changes only influence the hinting process; an outline’s direction won’t be changed at all.

2.3.17 Miscellaneous

Watch input files (GUI only)

If this checkbox is set, automatically regenerate the output file as soon as an input file (either the font or the control instructions file) gets modified.

Pressing the ‘Run’ button starts watching. If an error occurs, watching stops and must be restarted with the ‘Run’ button.

--ignore-restrictions, -i

By default, fonts that 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.

--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 that 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 in general.

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

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.

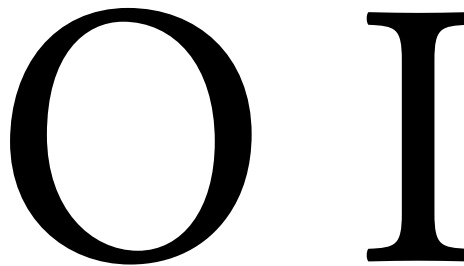


Figure 3.1: The letter ‘O’ has two contours, an inner and an outer one, while letter ‘I’ has only an outer contour.

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

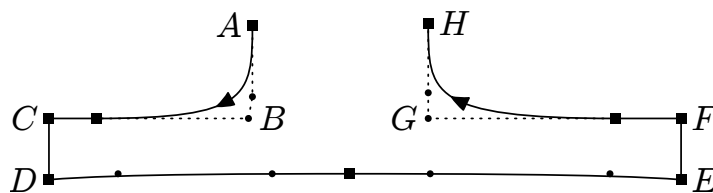


Figure 3.2: A serif. Contour and control points are represented by squares and circles, respectively. The bottom ‘line’ DE is approximately aligned along the horizontal axis, thus it forms a segment of 7 points. Together with the two other horizontal segments, BC and FG, they form two edges (BC+FG, DE).

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 writing system, see below ([section 3.8](#)).

3.2 Feature Analysis

The auto-hinter analyzes a font in two steps. Right now, everything described here happens for the horizontal axis only, providing vertical hinting.

- Global Analysis

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

- Compute standard horizontal stem width of the font. The value is normally taken from glyphs that resemble letter ‘o’.
- Compute blue zones, see below ([section 3.3](#)).

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

Script	Standard characters
cyr1	‘o’, U+043E, CYRILLIC SMALL LETTER O
	‘O’, U+041E, CYRILLIC CAPITAL LETTER O
grek	‘o’, U+03BF, GREEK SMALL LETTER OMICRON
	‘O’, U+039F, GREEK CAPITAL LETTER OMICRON
hebr	‘𐤌’, U+05DD, HEBREW LETTER FINAL MEM
latn	‘o’, U+006F, LATIN SMALL LETTER O
	‘O’, U+004F, LATIN CAPITAL LETTER O
	‘0’, U+0030, DIGIT ZERO

Table 3.1: script-specific standard characters of the ‘latin’ writing system

- 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

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

Here a series of tables that show the blue zone characters of the latin writing system’s available scripts; the values are hard-coded in the source code. Since the auto-hinter takes mean values it is not necessary that all characters of a zone are present.

ID	Blue zone	Characters
1	top of capital letters	THEZOCQS

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

Table 3.2: latn blue zones

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 2.3.5](#)).

ID	Blue zone	Characters
1	top of capital letters	ΓΒΕΖΘΟΩ
2	bottom of capital letters	ΒΛΖΞΘΟ
3	top of ‘small beta’ like letters	βθδζλξ
4	top of small letters	αειοπστω
5	bottom of small letters	αειοπστω
6	bottom of descenders of small letters	βγημρφχψ

Table 3.3: grek blue zones

ID	Blue zone	Characters
1	top of capital letters	БВЕПЗОСЭ
2	bottom of capital letters	БВЕПЗОСЭ
3	top of small letters	хпншезос
4	bottom of small letters	хпншезос
5	bottom of descenders of small letters	pyφ

Table 3.4: cyr1 blue zones

ID	Blue zone	Characters
1	top of letters	סמכךהדב
2	bottom of letters	צסמכטב
3	bottom of descenders of letters	זףיוךק

Table 3.5: hebr blue zones

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.

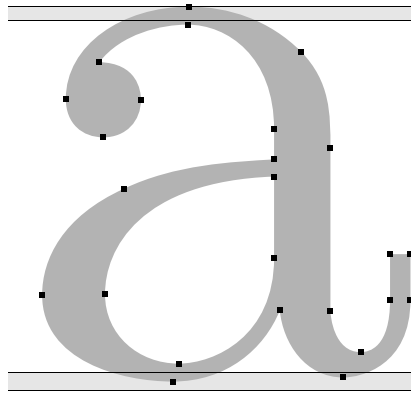


Figure 3.3: Two blue zones relevant to the glyph ‘a’. Vertical point coordinates of *all* glyphs within these zones are aligned, provided the blue zone is active (this is, its vertical size is smaller than 3/4 pixels).

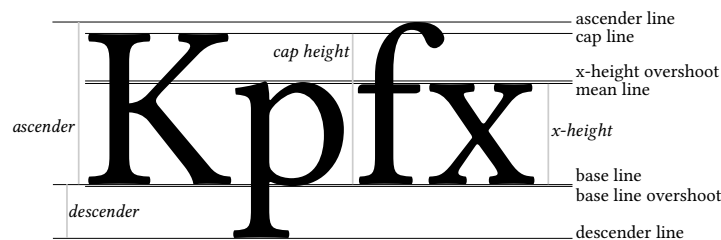


Figure 3.4: This image shows the relevant glyph terms for vertical blue zone positions.

This is the central routine of the auto-hinter; its actions are highly dependent on the used writing system. Currently, only one writing system is available (latin), providing support for scripts like Latin or Greek.

- 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**.

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 whether a

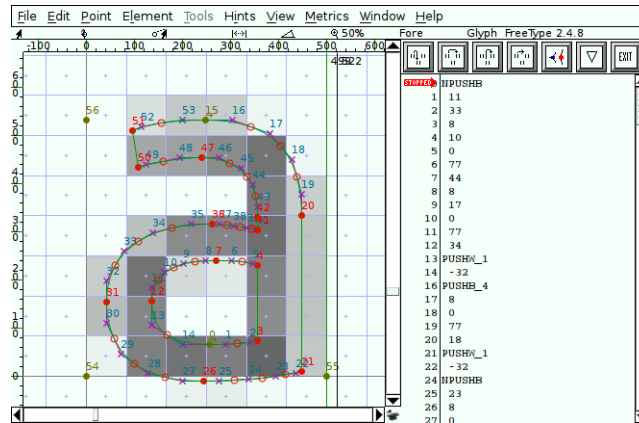


Figure 3.5: Before hinting.

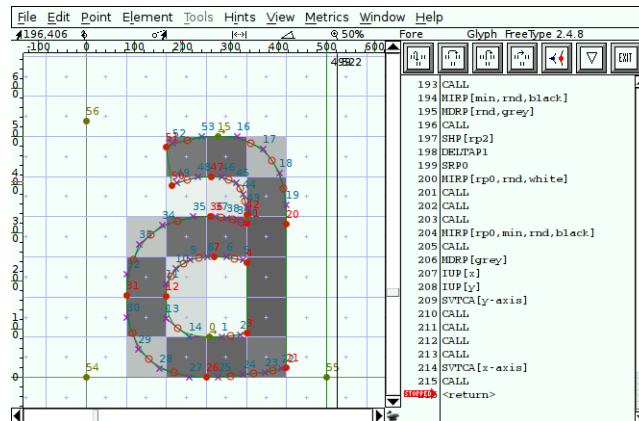


Figure 3.6: After hinting, using manual hints.

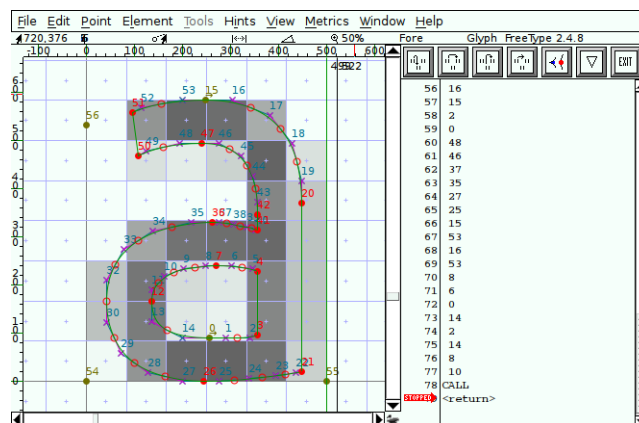


Figure 3.7: After hinting, using ttfautohint. Note that the hinting process doesn't change horizontal positions.

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 ‘`afi10108`’) from the ‘Ubuntu book’ font, processed with `ttfautohint`.

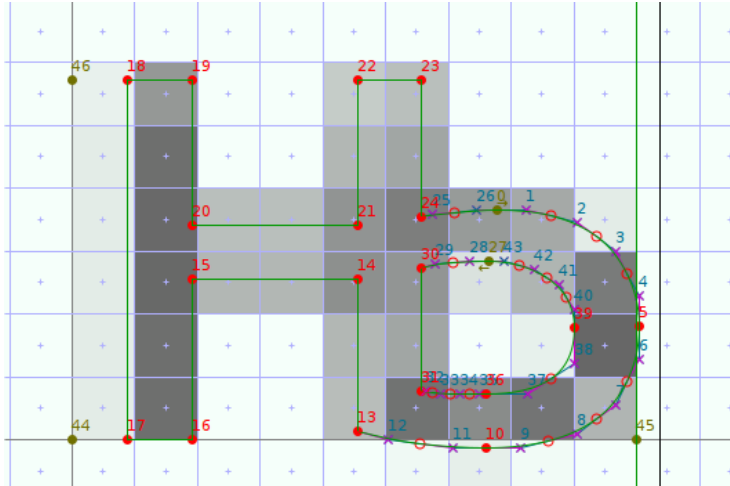


Figure 3.8: Before hinting, size 11px.

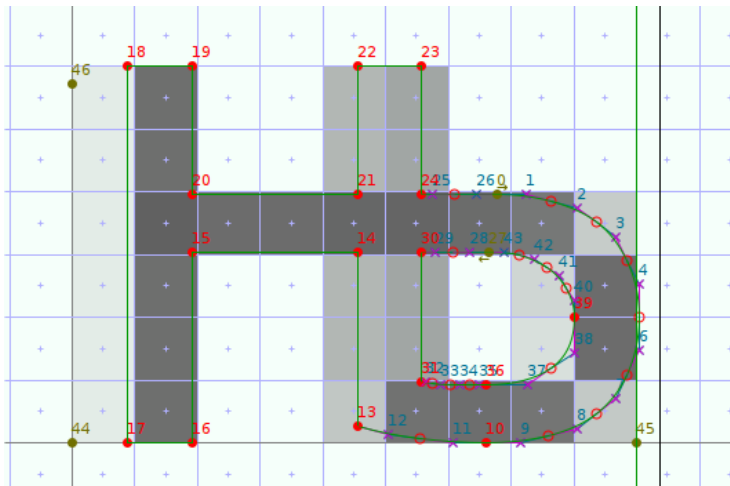


Figure 3.9: After hinting, size 11px. Segments 43-27-28 and 14-15 are aligned on a single edge, as are segments 26-0-1 and 20-21.

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

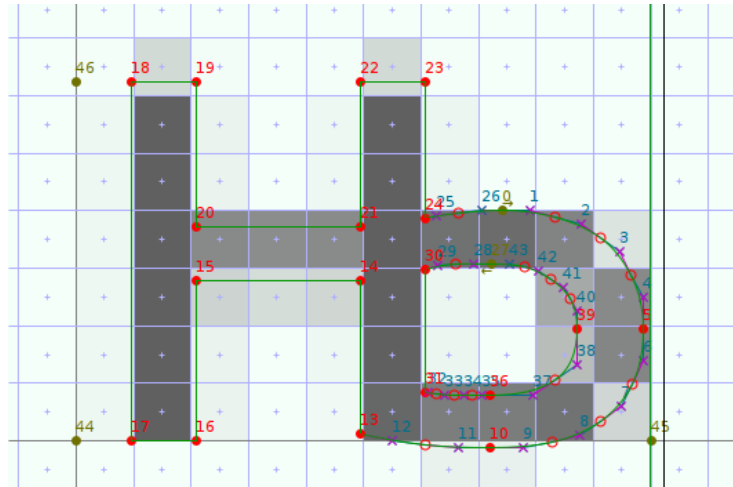


Figure 3.10: Before hinting, size 12px.

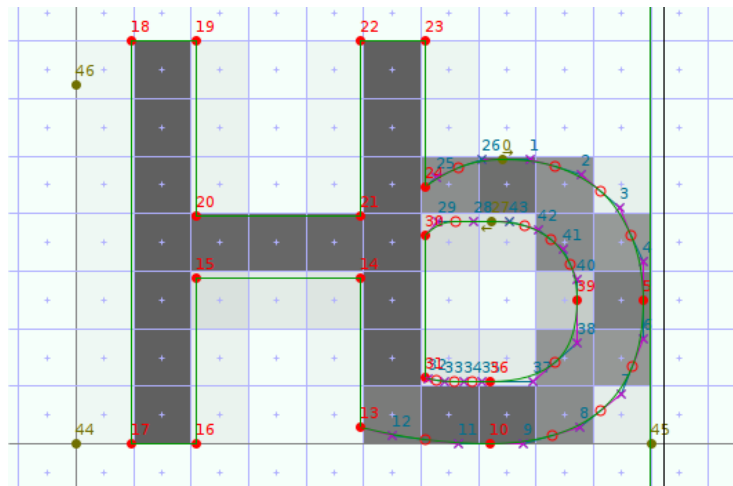


Figure 3.11: After hinting, size 12px. The segments are not aligned. While segments 43-27-28 and 20-21 now have almost the same horizontal position, they don't form an edge because the outlines passing through the segments point into different directions.

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 $10 \leq \text{ppem} \leq 100$, 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 this hinting range: 10-15, 16-40, 41-80, and 81-100. For `ppem` values below 10 `ppem`, the hint set covering 10-15 `ppem` is used, for `ppem` values larger than 100 the hint set covering 81-100 `ppem` is used. For `ppem` values larger than 250, no hinting gets applied.

3.6 Composite Glyphs

The `ttfautohint` library (and programs) supports two solutions for handling composite glyphs, to be controlled with option `--composites` ([subsection 2.3.10](#)). This section contains some general information, then covers the case where the option is off, while the next section describes how `ttfautohint` behaves if this option is activated.

Regardless of the `--composites` option, `ttfautohint` performs a scan over all composite glyphs to assure that components of a composite glyph inherit its style, as described later ([section 3.10](#)). However, components that are shifted vertically will be skipped. For example, if the glyph 'Agrave' uses a shifted 'grave' accent glyph, the accent is ignored. On the other hand, if there is a glyph 'agrave' that uses the same 'grave' glyph vertically unshifted, 'agrave' does inherit the style.

If `--composites` is off, components are hinted separately, then put together. Separate hinting implies that the current style's blue zones are applied to all subglyphs in its original, unshifted positions.

3.7 The '.ttfautohint' Glyph

If option `--composites` ([subsection 2.3.10](#)) is used, `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 components: '.ttfautohint', 'A', and 'grave' (in this order).

Bytecode of	Action
.ttfautohint	increase <code>cvtl_is_subglyph</code> (now: 1)

Bytecode of	Action
A	do nothing because <code>cvtl_is_subglyph > 0</code>
grave	do nothing because <code>cvtl_is_subglyph > 0</code>
Agrave	decrease <code>cvtl_is_subglyph</code> (now: 0)
	apply hints because <code>cvtl_is_subglyph == 0</code>

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 that are able to bypass the cmap table might be affected.)

3.8 Writing Systems

In FreeType terminology, a writing system is a set of functions that provides auto-hinting for certain scripts. Right now, only two writing systems from FreeType’s auto-hinter are available in `ttfautohint`: ‘dummy’ and ‘latin’. The former handles the ‘no-script’ case; details to ‘latin’ follow in the next section.

3.9 Scripts

`ttfautohint` needs to know which script 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.

Here is the hardcoded list of character ranges that are used for scripts in the ‘latin’ writing system. As you can see, this also covers some non-latin scripts (in the Unicode sense) that have similar typographical properties.

In `ttfautohint`, scripts are identified by four-character tags. The value `none` indicates ‘no script’.

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
0x1D00 - 0x1D7F	Phonetic Extensions
0x1D80 - 0x1DBF	Phonetic Extensions Supplement
0x1DC0 - 0x1DFF	Combining Diacritical Marks Supplement
0x1E00 - 0x1EFF	Latin Extended Additional

Character range	Description
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
0x2E00 - 0x2E7F	Supplemental Punctuation
0xA720 - 0xA7FF	Latin Extended-D
0xFB00 - 0xFB06	Alphabetical Presentation Forms (Latin Ligatures)
0x1D400 - 0x1D7FF	Mathematical Alphanumeric Symbols
0x1F100 - 0x1F1FF	Enclosed Alphanumeric Supplement

Table 3.7: latin character ranges

Character range	Description
0x0370 - 0x03FF	Greek and Coptic
0x1F00 - 0x1FFF	Greek Extended

Table 3.8: greek character ranges

Character range	Description
0x0400 - 0x04FF	Cyrillic
0x0500 - 0x052F	Cyrillic Supplement
0x2DE0 - 0x2DFF	Cyrillic Extended-A
0xA640 - 0xA69F	Cyrillic Extended-B

Table 3.9: cyril character ranges

Character range	Description
0x0590 - 0x05FF	Hebrew
0xFB1D - 0xFB4F	Alphabetic Presentation Forms (Hebrew)

Table 3.10: hebrew character ranges

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 that essentially does nothing). This can be changed by specifying a *fallback script*; see option `--fallback-script` ([subsection 2.3.3](#)).

3.10 OpenType Features

(Please read the [OpenType specification](#) for details on *features*, GSUB, and GPOS tables, and how they relate to scripts.)

For modern OpenType fonts, character ranges are not sufficient to handle scripts.

- Due to glyph substitution in the font (as specified in a font’s GSUB table), which handles ligatures and similar typographic features, there is no longer a one-to-one mapping from an input Unicode character to a glyph index. Some ligatures, like ‘fi’, actually do have Unicode values for historical reasons, but most of them don’t. While it is possible to map ligature glyphs into Unicode’s Private Use Area (PUA), code values from this area are arbitrary by definition and thus unusable for `ttfautohint`.
- Some features like `sup`s (for handling superscript) completely change the appearance and even vertical position of the affected glyphs. Obviously, the blue zones for ‘normal’ glyphs no longer fit, thus the auto-hinter puts them into a separate group (called *style* in FreeType speak), having its own set of blue zones.

Feature tag	Description
<code>c2cp</code>	petite capitals from capitals
<code>c2sc</code>	small capitals from capitals
<code>ordn</code>	ordinals
<code>pcap</code>	petite capitals
<code>sinf</code>	scientific inferiors
<code>smcp</code>	small capitals
<code>subs</code>	subscript
<code>sup</code> s	superscript
<code>titl</code>	titling

Table 3.11: OpenType features handled specially by `ttfautohint`

There are two conditions to get a valid style for a feature in a given script.

1. One of the script’s standard characters must be available in the feature.
2. The feature must provide characters to form at least one blue zone; see above ([section 3.3](#)).

An additional complication is that features from the above table might use data not only from the GSUB but also from the GPOS table, containing information for glyph positioning. For example, the `sup`s feature for superscripts might use the same glyphs as the `subs` feature for subscripts, simply moved up. `ttfautohint` skips such vertically shifted glyphs (except for accessing standard characters) because glyph positioning happens after hinting. Continuing our example, the `sup`s feature wouldn’t form a style, contrary to `subs`, which holds the unshifted glyphs.

The remaining OpenType features of a script are not handled specially; the affected glyphs are simply hinted together with the ‘normal’ glyphs of the script.

Note that a font might still contain some features not covered yet: OpenType has the concept of a *default script*; its data gets used for all scripts that aren’t explicitly handled in a font. By default, `ttfautohint`

unifies all affected glyphs from default script features with the `latn` script. This can be changed with option `--default-script` ([subsection 2.3.2](#)), if necessary.

`ttfautohint` uses the [HarfBuzz](#) library for handling OpenType features.

3.11 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`: All hints in the table are replaced with new ones. If option `--composites` ([subsection 2.3.10](#)) is used, one glyph gets added (namely the `'ttfautohint'` glyph) and all composites get an additional component.
- `cvt`, `prep`, and `fpgm`: These tables get replaced with data necessary for the new hinting bytecode.
- `gasp`: Set up to always use grayscale rendering, for all sizes, 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 that have been used for calling `ttfautohint`. This can be controlled with the `--no-info` ([subsection 2.3.13](#)) 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.

3.12 Problems

3.12.1 Interaction With FreeType

Recent versions of FreeType have an experimental extension for handling subpixel hinting; it is off by default and can be activated by defining the macro `TT_CONFIG_OPTION_SUBPIXEL_HINTING` at compile time. This code has been contributed mainly by [Infinality](#), being a subset of his original patch. Many GNU/Linux distributions activate this code, or provide packages to activate it.

This extension changes the behaviour of many bytecode instructions to get better rendering results. However, not all changes are global; some of them are specific to certain fonts. For example, it contains font-specific improvements for the `'DejaVu Sans'` font family. The list of affected fonts is hard-coded; it can be found in FreeType's source code file `ttsubpix.c`.

If you are going to process such specially-handled fonts with `ttfautohint`, serious rendering problems might show up. Since `ttfautohint` (intentionally) doesn't change the font name in the name table, the

Infinality extension has no chance to recognize that the hints are different. All such problems vanish if the font gets renamed in its name table (the name of the font file itself doesn't matter).

3.12.2 Incorrect Unicode Character Map

Fonts with an incorrect Unicode cmap table will not be properly hinted by ttfautohint. Especially older fonts do cheat; for example, there exist Hebrew fonts that map its glyphs to character codes 'A', 'B', etc., to make them work with non-localized versions of Windows 98, say.

Since ttfautohint needs to find both standard and blue zone characters, it relies on correct Unicode values. If you want to handle such fonts, please fix their cmap tables accordingly.

3.12.3 Irregular Glyph Heights

The central concept of ttfautohint's hinting algorithm, as discussed above ([section 3.1](#)), is to identify horizontal segments at extremum positions, especially for blue zones. If such a segment is missing, it cannot be associated with a blue zone, possibly leading to irregular heights for the particular glyph.

Normally, a segment has a horizontal length of at least 20 font units (assuming 2048 units per EM)¹. Using a Control Instructions File ([subsection 2.3.16](#)), however, it is possible to define additional segments at arbitrary points that help overcome this restriction, making it possible to fix (most) of such problems.

3.12.4 Diagonals

ttfautohint doesn't handle diagonal lines specially. For thin outlines, this might lead to strokes that look too thick at smaller sizes. A font designer might compensate this to a certain amount by slightly reducing the stroke width of diagonal lines. However, in many cases the sub-optimal appearance of a stroke with borders that don't exactly fit the pixel grid is not the outline itself but an incorrect gamma value of the monitor: People tend to not properly adjust it, and the default values of most operating systems are too low, causing too much darkening of such strokes. It is thus of vital importance to compare ttfautohint's results with similar fonts to exclude any systematic effect not related to the outlines themselves.

¹To be more precise, the sum of the height and width of a segment must be at least 20 font units, and the height multiplied by 14 must not exceed the length. Thus (19,1) is also a valid minimum (length,height) pair, while (18,2) isn't. The value 20 is heuristic and hard-coded, as is the value 14 (corresponding to a slope of approx. 4.1°).

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_Error_Func

A callback function to get error information.

error is the value TTF_autohint returns. See file `ttfautohint-errors.h` for a list. Error codes not in this list are directly taken from FreeType; see the FreeType header file `fterrdef.h` for more.

error_string, if non-NULL, is a pointer to an error message that represents *error*.

The next three parameters help identify the origin of text string parsing errors. *linenum*, if non-zero, contains the line number. *line*, if non-NULL, is a pointer to the input line that can't be processed. *errpos*, if non-NULL, holds a pointer to the position in *line* where the problem occurs.

error_data is a void pointer to user-supplied data.

```
typedef void
(*TA_Error_Func)(TA_Error error,
                  const char* error_string,
                  unsigned int linenum,
                  const char* line,
                  const char* errpos,
                  void* error_data);
```

4.4 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 of the name table](#) 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);
```

4.5 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 that 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.

4.5.1 I/O

`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 that 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 that 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`.

`control-file`

A pointer of type `FILE*` to the data stream of control instructions. Mutually exclusive with `control-buffer`.

An entry in a control instructions file or buffer has one of the following syntax forms:

```
[ font-idx ] glyph-id l|r points [ ( left-offset , right-offset ) ]  
[ font-idx ] glyph-id n points  
[ font-idx ] glyph-id p points [ x x-shift ] [ y y-shift ] @ ppems
```

font-idx gives the index of the font in a TrueType Collection. If missing, it is set to zero. For normal TrueType fonts, only value zero is valid. If starting with `0x` the number is interpreted as hexadecimal. If starting with `0` it gets interpreted as an octal value, and as a decimal value otherwise.

glyph-id is a glyph's name as listed in the post SFNT table or a glyph index. A glyph name consists of characters from the set 'A-Za-z0-9._' only and does not start with a digit or period, with the

exceptions of the names `‘.notdef’` and `‘.null’`. A glyph index can be specified in decimal, octal, or hexadecimal format, the latter two indicated by the prefixes 0 and 0x, respectively.

The mutually exclusive parameters `‘l’` and `‘r’` indicate that the following points have left or right ‘out’ direction, respectively, overriding `ttfautohint`’s algorithm for setting point directions. The ‘out direction’ of a point is the direction of the outline *leaving* the point (or passing the control point). If the specified direction is identical to what `ttfautohint` computes, nothing special happens. Otherwise, a one-point segment with the specified direction gets created. By default, its length is zero. Setting *left-offset* and *right-offset*, you can change the segment’s horizontal start and end position relative to the point position. *left-offset* and *right-offset* are integers measured in font units.

Parameter `‘n’` sets the ‘out’ direction of the following points to ‘no direction’. If the specified direction is identical to what `ttfautohint` computes, nothing special happens. Otherwise, `ttfautohint` no longer considers those points as part of horizontal segments, thus treating them as ‘weak’ points.

Modifying or adding segments don’t directly modify the outlines; it only influences the hinting process.

Parameter `‘p’` makes `ttfautohint` apply delta exceptions for the given points, shifting the points by the given values. Note that those delta exceptions are applied *after* the final IUP instructions in the bytecode; as a consequence, they are (partially) ignored by rasterizers if in ClearType mode.

Both *points* and *ppems* are number ranges, similar to the `x-height-snapping-exceptions` syntax.

x-shift and *y-shift* represent real numbers that get rounded to multiples of 1/8 pixels. The entries for `‘x’` and `‘y’` are optional; if missing, the corresponding value is set to zero.

Values for *x-shift* and *y-shift* must be in the range `[-1.0;1.0]`. Values for *ppems* must be in the range `[6;53]`. Values for *points* are limited by the number of points in the glyph.

Control instruction entries can be either separated with newlines or with character `‘;’`. Additionally, a line can be continued on the next line by ending it with backslash character (`‘\’`). A backslash followed by a newline gets treated similar to a whitespace character.

A comment starts with character `‘#’`; the rest of the line is ignored. An empty line is ignored also.

Note that only character `‘.’` is recognized as a decimal point, and a thousands separator is not accepted.

`control-buffer`

A pointer of type `const char*` to a buffer that contains control instructions. Needs `control-buffer-len`. Mutually exclusive with `control-file`.

`control-buffer-len`

A value of type `size_t`, giving the length of the control instructions buffer. Needs `control-buffer`.

4.5.2 Messages and Callbacks

`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 that is passed to the progress callback function.

error-string

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

error-callback

A pointer of type `TA_Error_Func` ([section 4.3](#)), specifying a callback function for error messages. This function gets called right before `TTF_autohint` exits. If this field is not set or set to NULL, no error callback function is used.

Use it as a more sophisticated alternative to `error-string`.

error-callback-data

A point of type `void*` to user data that is passed to the error callback function.

info-callback

A pointer of type `TA_Info_Func` ([section 4.4](#)), 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 that is passed to the info callback function.

debug

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

4.5.3 General Hinting Options

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`) that 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.

hint-composites

If this integer is set to 1, composite glyphs get separate hints. This implies adding a special glyph to the font called `'ttfautohint'` ([section 3.7](#)). Setting it to 0 (which is the default), the hints of the composite glyphs' components are used. Adding hints for composite glyphs increases the size of the resulting bytecode a lot, but it might deliver better hinting results. However, this depends on the processed font and must be checked by inspection.

`adjust-subglyphs`

An integer (1 for ‘on’ and 0 for ‘off’, which is the default) to specify whether native TrueType hinting of the *input font* 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 only if the old hints move or scale subglyphs independently of the output resolution, for example some exotic CJK fonts.

`pre-hinting` is a deprecated alias name for this option.

4.5.4 Hinting Algorithms

`gray-strong-stem-width`

An integer (1 for ‘on’ and 0 for ‘off’, which is the default) that 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’) that 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 $36 \leq \text{version} < 38$ and ClearType is enabled.

`dw-cleartype-strong-stem-width`

An integer (1 for ‘on’ and 0 for ‘off’, which is the default) that 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 $6 \leq \text{PPEM} \leq \text{increase-x-height}$, round up the font’s x height much more often than normally (to use the terminology of TrueType’s ‘Super Round’ bytecode instruction, the threshold gets increased from 5/8px to 13/16px). 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.

`x-height-snapping-exceptions`

A pointer of type `const char*` to a null-terminated string that 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 *value*₁-*value*₂, meaning $\text{value}_1 \leq \text{PPEM} \leq \text{value}_2$. *value*₁ or *value*₂ (or both) can be missing; a missing value is replaced by the beginning or end of the whole interval of valid PPEM values, respectively. Whitespace is not significant; superfluous commas are ignored, and ranges must be specified in increasing order. For example, the string “3, 5-7, 9-” means the values 3, 5, 6, 7, 9, 10, 11, 12, etc. Consequently, if the supplied argument is “-”, no x height snapping takes place at all. The default is the empty string (“”), meaning no snapping exceptions.

`windows-compatibility`

If this integer is set to 1, two artificial blue zones are used, positioned at the `usWinAscent` and `usWinDescent` values (from the font’s OS/2 table). The idea is to help `ttfautohint` so that the hinted glyphs stay within this horizontal stripe since Windows clips everything falling outside. The default is 0.

4.5.5 Scripts

`default-script`

A string consisting of four lowercase characters that specifies the default script for OpenType features. After applying all features that are handled specially, use this value for the remaining features. The default value is "latn"; if set to "none", no script is used. Valid values can be found in the header file `ttfautohint-scripts.h`.

`fallback-script`

A string consisting of four lowercase characters that specifies the default script for glyphs that can't be mapped to a script automatically. If set to "none" (which is the default), no script is used. Valid values can be found in the header file `ttfautohint-scripts.h`.

`symbol`

Set this integer to 1 if you want to process a font that `ttfautohint` would refuse otherwise because it can't find a single standard character for any of the supported scripts. `ttfautohint` then uses a default (hinting) value for the standard stem width instead of deriving it from a script's set of standard characters (for the latin script, one of them is character 'o'). The default value of this option is 0.

`fallback-stem-width`

Set the horizontal stem width (hinting) value for all scripts that lack proper standard characters. The value is given in font units and must be a positive integer. If not set, or the value is zero, `ttfautohint` uses a hard-coded default (50 units at 2048 units per EM, and linearly scaled for other UPEM values, for example 24 units at 1000 UPEM).

For symbol fonts (i.e., option `symbol` is given), `fallback-stem-width` has an effect only if `fallback-script` is set also.

4.5.6 Miscellaneous

`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.

`TTFA-info`

If set to 1, `ttfautohint` creates an SFNT table called TTFA and fills it with information on the parameters used while calling `TTF_autohint`. The format of the output data resembles the information at the very beginning of the dump emitted by option `debug`. The default value is 0.

Main use of this option is for font editing purposes. For example, after a font editor has added some glyphs, a front-end to `TTF_autohint` can parse TTFA and feed the parameters into another call of `TTF_autohint`. The new glyphs are then hinted while hints of the old glyphs stay unchanged.

If this option is not set, and the font to be processed contains a TTFA table, it gets removed.

Note that such a TTFA table gets ignored by all font rendering engines. In TrueType Collections, the TTFA table is added to the first subfont.

`dehint`

If set to 1, remove all hints from the font. All other hinting options are ignored.

4.5.7 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 process its own output a second time only if option `hint-composites` is not set (or if the font doesn't contain composite glyphs at all). This limitation might change in the future.

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 generic instructions how to compile the `ttfautohint` library together with its front-ends using a POSIX compatible shell and compiler.

5.1 Unix-like Platforms

The generic instructions should work just fine. Since `ttfautohint` depends on `Qt` version 4.x.x, `FreeType` version 2.4.5 or newer, and `HarfBuzz` version 0.9.19 or newer, you should install packages for these libraries (called `'libqt4'`, `'libfreetype6'`¹, and `'libharfbuzz0'` or similar) together with its development bundles (called `'libqt4-devel'`, `'freetype2-devel'`, and `'harfbuzz-devel'` or similar) before running `ttfautohint`'s `configure` script.

5.2 MS Windows

Precompiled binaries `ttfautohint.exe` and `ttfautohintGUI.exe` are available, being statically linked to `Qt`, `FreeType`, and `HarfBuzz`. This means that the two programs are not dependent on any other program-specific DLL, and you can move them to any place you like.

Hints for compilation with the `MinGW` environment are given in `INSTALL.git`.

5.3 Mac OS X

Right now, only a precompiled binary `ttfautohint` is offered; a ready-to-run app bundle for the GUI version is not yet available.

Detailed instructions to compile both `ttfautohint` and `ttfautohintGUI` can be found on [ttfautohint's homepage](#).

¹The number '6' indicates the version of the shared library of FreeType, which is not directly related to the source code version of FreeType.

6 Authors

Copyright © 2011-2014 by **Werner Lemberg**.

Copyright © 2011-2014 by **Dave Crossland**.

This file is part of the ttfautohint library, and may only be used, modified, and distributed under the terms given in **COPYING**. By continuing to use, modify, or distribute this file you indicate that you have read **COPYING** and understand and accept it fully.

The file **COPYING** mentioned in the previous paragraph is distributed with the ttfautohint library.

7 History

7.1 New in Version 1.2 (2014-Oct-06)

- It is now possible to control the hinting process at a very low level using a ‘control instructions’ file. Right now, two modes are supported:
 - Adding segments and changing segment directions.
 - Applying delta exceptions. Note that this doesn’t really work in ClearType.

Please see the documentation for a description and a small tutorial.

In the TTY front-end, use option `--control-file` to load such a file.

- Support for input file watching in `ttfautohintGUI`: If the ‘Watch Input File’ box is checked, the program automatically regenerates the output file as soon as the input font or control instructions file gets modified.

The idea is to dock `ttfautohintGUI` to your favourite font and text editor instead of adding a cheap text editor to `ttfautohintGUI` itself.

- With the new option `--fallback-stem-width` it is now possible to set a default stem width for all scripts that lack proper standard characters in the font.
- Add alias `--adjust-subglyphs` for option `--pre-hinting` to better describe its functionality. The short form `-p` stays unchanged.
- New option `--ttfa-table` to add an SFNT table TTFA to the output font, containing an ASCII dump of all used `ttfautohint` parameters (including control instructions). Right now, this is mainly for archiving purposes. Forthcoming versions of `ttfautohint` will be able to re-use this data if a font gets re-processed.
- A harmless buglet was fixed that made the `glyf` table too large by one byte under some circumstances.
- A bug fix taken from FreeType, correcting a crash for unusual outlines.
- Better handling of TTC files.
- New library options `error-callback` and `error-callback-data` for improved diagnostics. [No change in the front-ends except better error messages.]
- Many other, minor fixes and improvements.

7.2 New in Version 1.1 (2014-May-05)

- Support for the Devanagari script.

- Bug fixes in the computation of strong stem widths. GDI hinting now gives much more consistent results.
- Better recognition of script coverage: ttfautohint now properly scans composite glyphs to cover components also.
- Improved glyph shape analysis: Non-flat local extrema are now recognized more reliably, and stem detection works better.

7.3 New in Version 1.00 (2014-Mar-20)

- Much less memory consumption while handling fonts with complicated glyphs.
- Option `-s` was partially broken.

7.4 New in Version 1.00rc1 (2014-Feb-07)

- OpenType feature support. ttfautohint now uses the HarfBuzz library to analyze data from the GSUB table. This allows the hinting of glyphs that don't have an entry in a font's cmap table, for example superscripts or small caps.

Related to this, the new option `--default-script` controls the default (fallback) script used for OpenType features.

- More than a single standard character is used. For example, the 'latin' script uses characters 'o', 'O', and digit '0'. This improves the hinting of fonts (and features) that have only a partial coverage of a script's character repertoire.
- Much better GDI ClearType hinting in the range 30-80ppem (approx.), avoiding overly flat tops and bottoms of round glyphs.
- Better handling of non-square pixels (this is, horizontal and vertical resolutions differ) in the created TrueType bytecode.

7.5 New in Version 0.97 (2013-Nov-09)

- Improved script support. Besides Cyrillic and Greek, which are now handled separately from Latin, ttfautohint can handle Hebrew.
- Option `-f` now takes a parameter to specify the fallback script. The corresponding long option name has been renamed from `--latin-fallback` to `--fallback-script`.
- Work around a bug in display environments that use FreeType 2.5.0 and earlier for rendering: Sometimes, the 'strong' stem width routine was used for DW ClearType (this is, subpixel hinting in FreeType is enabled) even if 'smooth' was selected while generating the font with ttfautohint.

7.6 New in Version 0.96 (2013-Aug-06)

- Option `--components` has been replaced with `--composites`: By default, the components of a composite glyph are now hinted separately, since tests has shown that this gives good results in most cases. If this option is set, however, the composite glyph itself gets hinted (and the hints of the components are ignored).

An unfortunate side effect is that `ttfautohint`'s option `-c` (which stays as a shorthand for `--composites`) now does exactly the opposite as in previous releases.

- Older versions of Monotype's 'iType' bytecode interpreter have a serious bug: The DIV instruction rounds the result, while the correct operation is truncation. This caused 'exploding characters' with fonts hinted by `ttfautohint`. Since many printers contain this rasterizer without any possibility to update to a non-buggy version, `ttfautohint` now contains work-arounds to circumvent the problem.
- Better support for glyphs where some points have almost the same position (for example glyph 'Oslash' in font 'Roboto-Thin').
- Better support for glyphs that use explicit 'on' points around round extrema.

7.7 New in Version 0.95 (2013-Mar-07)

- New option `--dehint` to strip off all hints without generating new hints. This option is intended for testing purposes.
- Minor fixes to the created bytecode for compatibility.
- Minor GUI improvements.

7.8 New in Version 0.94 (2012-Nov-29)

- New option `--windows-compatibility` that adds two artificial blue zones at vertical positions given by 'usWinAscent' and 'usWinDescent'. This helps `ttfautohint`'s hinting algorithm reduce the possibility of clipping if those two values are very tight.
- Implement option `--x-height-snapping-exceptions`, making `ttfautohint` avoid x-height snapping for selected PPEM values. Useful in combination with `--windows-compatibility`.
- Minor fixes to the created bytecode for compatibility and robustness.

7.9 New in Version 0.93 (2012-Oct-09)

- New option `--components` to treat components of composite glyphs separately. This greatly reduces the bytecode size.

I'm waiting for reports whether this option works for most fonts; in case this is true I'm inverting the option, making it the default (and the old behaviour optional).

- Full support of TTCs, this is, all subfonts get auto-hinted now.

- The upper limit of the `--increase-x-height` option has been removed.
- Drag-and-drop support in the GUI.
- The TTY version of `ttfautohint` now acts like a (Unix) filter, this is, it accepts stdin and stdout as input and output, respectively.
- Less memory consumption.

7.10 New in Version 0.92 (2012-Aug-07)

- A serious bug in the created bytecode has been fixed, causing incorrect rounding.

7.11 New in Version 0.91 (2012-Jul-12)

- A new, ‘strong’ routine to handle stem widths and positions has been added, to be selected with the `--strong-stem-width` command line option. If it is active, stem widths and positions are snapped to the grid as much as possible. This algorithm is useful for GDI ClearType support.
- A new command line option `--debug` (not available for `ttfautohintGUI`) to print very detailed debugging information.

7.12 New in Version 0.9 (2012-Jun-06)

- The created bytecode has been reduced in size, making it approx. 20% smaller.
- New option `--symbol` to use standard stem height and width values instead of using character ‘o’ (which may be missing). Use this option for symbol fonts or math glyphs.
- More documentation (in text, HTML, and PDF format). It’s still incomplete, though.
- Option `--ignore-permissions` has been renamed to `--ignore-restrictions`. The short form is still `-i`.
- Defaults for various parameters have been set to more sensible values:
 - `hinting-range-max`: 50 (was 1000)
 - `hinting-limit`: 200 (was 1000)
- Option `--increase-x-height` now has a mandatory argument (in the range 6-20 or value 0 to disable it, default value is 14).

7.13 New in Version 0.8 (2012-Mar-21)

- Implement option `-x` to increase the x height of the font for small PPEM values by rounding up far more often then rounding down.
- Add option ‘-G n’ to switch off hinting completely above value n.
- `ttfautohint` now appends version information and the used parameters to the ‘Version’ field(s) in the ‘name’ table. This can be suppressed with option `-n`.

7.14 New in Version 0.7 (2012-Feb-05)

- A GUI has been added, using the Qt framework. The binary is called ‘ttfautohintGUI’.

7.15 New in Version 0.6.1 (2012-Jan-02)

- The improved handling of composite glyphs in 0.6 was buggy under certain circumstances, making ttfautohint crash and FontValidator complain.
- Dropout handling has been activated.

7.16 New in Version 0.6 (2011-Dec-25)

- Improved handling of composite glyphs.
- Implement option -p to pre-hint glyphs with original hints before conversion takes place.
- Don’t add a DSIG table if there is none in the input font.
- Human-readable error messages instead of hexadecimal error codes.
- Better tests (both at runtime and compile time) to reject too old FreeType versions.

7.17 New in Version 0.5 (2011-Nov-06)

- Rendering on iOS is now expected to give good results.
- No bad rendering at very large PPEM values.

7.18 New in Version 0.4 (2011-Oct-27)

- The bytecode has been changed to ‘create’ twilight points. This should avoid rendering artifacts on some platforms.

7.19 New in Version 0.3 (2011-Sep-09)

- Fix font generation; sometimes the glyf table was one byte too short, making the font invalid.

7.20 New in Version 0.2 (2011-Jul-19)

- Fix bytecode bugs that prevented correct rendering on some platforms.

7.21 New in Version 0.1 (2011-Jun-30)

- First release.