HarfBuzz Manual

This document is for HarfBuzz VERSION.

The current HarfBuzz codebase is versioned 2.x.x and is stable and under active maintenance. This is what is used in latest versions of Firefox, GNOME, ChromeOS, Chrome, LibreOffice, XeTeX, Android, and KDE, among other places.

Prior to 2012, the original HarfBuzz codebase (which, these days, is referred to as *harfbuzz-old*) was derived from code in FreeType, Pango, and Qt. It is *not* actively developed or maintained, and is extremely buggy. All users of harfbuzz-old are encouraged to switch over to the new HarfBuzz as soon as possible.

To make this distinction clearer in discussions, the current HarfBuzz codebase is sometimes referred to as *harfbuzz-ng*.

For reference purposes, the harfbuzz-old source tree is archived here. There are no release tarballs of harfbuzz-old whatsoever.

Core API

OpenType API

Apple Advanced Typography API

Integration API

What is HarfBuzz?

HarfBuzz is a *text-shaping engine*. If you give HarfBuzz a font and a string containing a sequence of Unicode codepoints, HarfBuzz selects and positions the corresponding glyphs from the font, applying all of the necessary layout rules and font features. HarfBuzz then returns the string to you in the form that is correctly arranged for the language and writing system.

HarfBuzz can properly shape all of the world's major writing systems. It runs on all major operating systems and software platforms and it supports the major font formats in use today.

What is text shaping?

Text shaping is the process of translating a string of character codes (such as Unicode codepoints) into a properly arranged sequence of glyphs that can be rendered onto a screen or into final output form for inclusion in a document.

The shaping process is dependent on the input string, the active font, the script (or writing system) that the string is in, and the language that the string is in.

Modern software systems generally only deal with strings in the Unicode encoding scheme (although legacy systems and documents may involve other encodings).

There are several font formats that a program might encounter, each of which has a set of standard text-shaping rules.

The dominant format is OpenType. The OpenType specification defines a series of shaping models for various scripts from around the world. These shaping models depend on the font incorporating certain features as *lookups* in its `GSUB` and `GPOS` tables.

Alternatively, OpenType fonts can include shaping features for the Graphite shaping model.

TrueType fonts can also include OpenType shaping features. Alternatively, TrueType fonts can also include Apple Advanced Typography (AAT) tables to implement shaping support. AAT fonts are generally only found on macOS and iOS systems.

Text strings will usually be tagged with a script and language tag that provide the context needed to perform text shaping correctly. The necessary script and language tags are defined by OpenType.

Why do I need a shaping engine?

Text shaping is an integral part of preparing text for display. Before a Unicode sequence can be rendered, the codepoints in the sequence must be mapped to the corresponding glyphs provided in the font, and those glyphs must be positioned correctly relative to each other. For many of the scripts supported in Unicode, these steps involve script-specific layout rules, including complex joining, reordering, and positioning behavior. Implementing these rules is the job of the shaping engine.

Text shaping is a fairly low-level operation. HarfBuzz is used directly by text-handling libraries like Pango, as well as by the layout engines in Firefox, LibreOffice, and Chromium. Unless you are *writing* one of these layout engines yourself, you will probably not need to use HarfBuzz: normally, a layout engine, toolkit, or other library will turn text into glyphs for you.

However, if you *are* writing a layout engine or graphics library yourself, then you will need to perform text shaping, and this is where HarfBuzz can help you.

Here are some specific scenarios where a text-shaping engine like HarfBuzz helps you:

- OpenType fonts contain a set of glyphs (that is, shapes to represent the letters, numbers, punctuation marks, and all other symbols), which are indexed by a `glyph ID`.

  A particular glyph ID within the font does not necessarily correlate to a predictable Unicode codepoint. For instance, some fonts have the letter ”a” as glyph ID 1, but many others do not. In order to retrieve the right glyph from the font to display ”a”, you need to consult the table inside the font (the `cmap` table) that maps Unicode codepoints to glyph IDs. In other words, *text shaping turns codepoints into glyph IDs*.

- Many OpenType fonts contain ligatures: combinations of characters that are rendered as a single unit. For instance, it is common for the ”f, i” letter sequence to appear in print as the single ligature glyph ”fi”.

  Whether you should render an ”f, i” sequence as `fi` or as ”fi” does not depend on the input text. Instead, it depends on the whether or not the font includes an ”fi” glyph and on the level of ligature application you wish to perform. The font and the amount of ligature application used are under your control. In other words, *text shaping involves querying the font's ligature tables and determining what substitutions should be made*.

- While ligatures like ”fi” are optional typographic refinements, some languages *require* certain substitutions to be made in order to display text correctly.

  For example, in Tamil, when the letter ”TTA” ( ) letter is followed by the vowel sign ”U” ( ), the pair must be replaced by the single glyph ” ”. The sequence of Unicode characters ” , ” needs to be substituted with a single ” ” glyph from the font.

  But ” ” does not have a Unicode codepoint. To find this glyph, you need to consult the table inside the font (the `GSUB` table) that contains substitution information. In other words, *text shaping chooses the correct glyph for a sequence of characters provided*.

- Similarly, each Arabic character has four different variants corresponding to the different positions it might appear in within a sequence. Inside a font, there will be separate glyphs for the initial, medial, final, and isolated forms of each letter, each at a different glyph ID.

  Unicode only assigns one codepoint per character, so a Unicode string will not tell you which glyph variant to use for each character. To decide, you need to analyze the whole string and determine the appropriate glyph for each character based on its position. In other words, *text shaping chooses* *the correct form of the letter by its position and returns the correct glyph from the font.*

- Other languages involve marks and accents that need to be rendered in specific positions relative a base character. For instance, the Moldovan language includes the Cyrillic letter "zhe" ( ) with a breve accent, like so: " ".

  Some fonts will provide this character as a single zhe-with-breve glyph, but other fonts will not and, instead, will expect the rendering engine to form the character by superimposing the separate " " and "˘" glyphs.

  But exactly where you should draw the breve depends on the height and width of the preceding zhe glyph. To find the right position, you need to consult the table inside the font (the `GPOS` table) that contains positioning information. In other words, *text shaping tells you whether you have a precomposed glyph within your font or if you need to compose a glyph yourself out of combining marksMDASHand, if so, where to position those marks.*

If tasks like these are something that you need to do, then you need a text shaping engine. You could use Uniscribe if you are writing Windows software; you could use CoreText on macOS; or you could use HarfBuzz.

In the rest of this manual, the text will assume that the reader is that implementor of a text-layout engine.

What does HarfBuzz do?

HarfBuzz provides text shaping through a cross-platform C API that accepts sequences of Unicode codepoints as input. Currently, the following OpenType shaping models are supported:

- Indic (covering Devanagari, Bengali, Gujarati, Gurmukhi, Kannada, Malayalam, Oriya, Tamil, Telugu, and Sinhala)
- Arabic (covering Arabic, N'Ko, Syriac, and Mongolian)
- Thai and Lao
- Khmer
- Myanmar
- Tibetan
- Hangul
- Hebrew
- The Universal Shaping Engine or *USE* (covering complex scripts not covered by the above shaping models)

- A default shaping model for non-complex scripts (covering Latin, Cyrillic, Greek, Armenian, Georgian, Tifinagh, and many others)
- Emoji (including emoji modifier sequences, flag sequences, and ZWJ sequences)

In addition to OpenType shaping, HarfBuzz supports the latest version of Graphite shaping (the ”Graphite 2” model) and AAT shaping.

HarfBuzz can read and understand TrueType fonts (.ttf), TrueType collections (.ttc), and OpenType fonts (.otf, including those fonts that contain TrueType-style outlines and those that contain PostScript CFF or CFF2 outlines).

HarfBuzz is designed and tested to run on top of the FreeType font renderer. It can run on Linux, Android, Windows, macOS, and iOS systems.

In addition to its core shaping functionality, HarfBuzz provides functions for accessing other font features, including optional GSUB and GPOS OpenType features, as well as all color-font formats (`CBDT`, `sbix`, `COLR/CPAL`, and `SVG-OT`) and OpenType variable fonts. HarfBuzz also includes a font-subsetting feature. HarfBuzz can perform some low-level math-shaping operations, although it does not currently perform full shaping for mathematical typesetting.

A suite of command-line utilities is also provided in the source-code tree, designed to help users test and debug HarfBuzz's features on real-world fonts and input.

What HarfBuzz doesn't do

HarfBuzz will take a Unicode string, shape it, and give you the information required to lay it out correctly on a single horizontal (or vertical) line using the font provided. That is the extent of HarfBuzz's responsibility.

It is important to note that if you are implementing a complete text-layout engine you may have other responsibilities that HarfBuzz will *not* help you with. For example:

- HarfBuzz won't help you with bidirectionality. If you want to lay out text that includes a mix of Hebrew and English, you will need to ensure that each buffer provided to HarfBuzz has all of its characters in the same order and that the directionality of the buffer is set correctly. This may mean segmenting the text before it is placed into HarfBuzz buffers. In other words, the user will hit the keys in the following sequence:

  ```
  A B C [space]      [space] D E F
  ```

  but will expect to see in the output:

  ```
  ABC    DEF
  ```

This reordering is called *bidi processing* (”bidi” is short for bidirectional), and there’s an algorithm as an annex to the Unicode Standard which tells you how to process a string of mixed directionality. Before sending your string to HarfBuzz, you may need to apply the bidi algorithm to it. Libraries such as ICU and fribidi can do this for you.

- HarfBuzz won’t help you with text that contains different font properties. For instance, if you have the string ”a *huge* breakfast”, and you expect ”huge” to be italic, then you will need to send three strings to HarfBuzz: `a`, in your Roman font; `huge` using your italic font; and `breakfast` using your Roman font again.

  Similarly, if you change the font, font size, script, language, or direction within your string, then you will need to shape each run independently and output them independently. HarfBuzz expects to shape a run of characters that all share the same properties.

- HarfBuzz won’t help you with line breaking, hyphenation, or justification. As mentioned above, HarfBuzz lays out the string along a *single line* of, notionally, infinite length. If you want to find out where the potential word, sentence and line break points are in your text, you could use the ICU library’s break iterator functions.

  HarfBuzz can tell you how wide a shaped piece of text is, which is useful input to a justification algorithm, but it knows nothing about paragraphs, lines or line lengths. Nor will it adjust the space between words to fit them proportionally into a line.

As a layout-engine implementor, HarfBuzz will help you with the interface between your text and your font, and that’s something that you’ll needMDASH-what you then do with the glyphs that your font returns is up to you.

Why is it called HarfBuzz?

HarfBuzz began its life as text-shaping code within the FreeType project (and you will see references to the FreeType authors within the source code copyright declarations), but was then extracted out to its own project. This project is maintained by Behdad Esfahbod, who named it HarfBuzz. Originally, it was a shaping engine for OpenType fontsMDASH”HarfBuzz” is the Persian for ”open type”.

Installing HarfBuzz

Downloading HarfBuzz

The HarfBuzz source code is hosted at github.com/harfbuzz/harfbuzz.

Tarball releases and Win32 binary bundles (which include the libharfbuzz DLL, hb-view.exe, hb-shape.exe, and all dependencies) of HarfBuzz can be downloaded from github.com/harfbuzz/harfbuzz/releases.

Release notes are posted with each new release to provide an overview of the changes. The project tracks bug reports and other issues on GitHub. Discussion and questions are welcome on the HarfBuzz mailing list.

The API included in the `hb.h` file will not change in a compatibility-breaking way in any release. However, other, peripheral headers are more likely to go through minor modifications. We will do our best to never change APIs in an incompatible way. We will *never* break the ABI.

Building HarfBuzz

Building on Linux

(1) To build HarfBuzz on Linux, you must first install the development packages for FreeType, Cairo, and GLib. The exact commands required for this step will vary depending on the Linux distribution you use.

For example, on an Ubuntu or Debian system, you would run:

```
sudo apt install gcc g++ libfreetype6-dev libglib2.0-dev libcairo2-dev
```

On Fedora, RHEL, CentOS, or other Red-HatNDASHbased systems, you would run:

```
sudo yum install gcc gcc-c++ freetype-devel glib2-devel cairo-devel
```

(2) The next step depends on whether you are building from the source in a downloaded release tarball or from the source directly from the git repository.

(2)(a) If you downloaded the HarfBuzz source code in a tarball, you can now extract the source.

From a shell in the top-level directory of the extracted source code, you can run `./configure` followed by `make` as with any other standard package.

This should leave you with a shared library in the `src/` directory, and a few utility programs including `hb-view` and `hb-shape` under the `util/` directory.

(2)(b) If you are building from the source in the HarfBuzz git repository, rather than installing from a downloaded tarball release, then you must install two more auxiliary tools before you can build for the first time: pkg-config and ragel.

On Ubuntu or Debian, run:

```
sudo apt-get install autoconf automake libtool pkg-config ragel gtk-doc-tools
```

On Fedora, RHEL, CentOS, run:

```
sudo yum install autoconf automake libtool pkgconfig ragel gtk-doc
```

With pkg-config and ragel installed, you can now run `./autogen.sh`, followed by `./configure` and `make` to build HarfBuzz.

Building on Windows

On Windows, consider using Microsoft's free vcpkg utility to build HarfBuzz, its dependencies, and other open-source libraries.

If you need to build HarfBuzz from source, first put the ragel binary on your `PATH`, then follow the appveyor CI cmake build instructions.

Building on macOS

There are two ways to build HarfBuzz on Mac systems: MacPorts and Homebrew. The process is similar to the process used on a Linux system.

(1) You must first install the development packages for FreeType, Cairo, and GLib. If you are using MacPorts, you should run:

```
sudo port install freetype glib2 cairo
```

If you are using Homebrew, you should run:

```
brew install freetype glib cairo
```

(2) The next step depends on whether you are building from the source in a downloaded release tarball or from the source directly from the git repository.

(2)(a) If you are installing HarfBuzz from a downloaded tarball release, extract the tarball and open a Terminal in the extracted source-code directory. Run:

```
./configure
```

followed by:

```
make
```

to build HarfBuzz.

(2)(b) Alternatively, if you are building HarfBuzz from the source in the HarfBuzz git repository, then you must install several built-time dependencies before proceeding.

If you are using MacPorts, you should run:

```
sudo port install autoconf automake libtool pkgconfig ragel gtk-doc
```

to install the build dependencies.

If you are using Homebrew, you should run:

```
brew install autoconf automake libtool pkgconfig ragel gtk-doc
```

Finally, you can run:

```
./autogen.sh
```

(3) You can now build HarfBuzz (on either a MacPorts or a Homebrew system) by running:

```
./configure
```

followed by:

```
make
```

This should leave you with a shared library in the `src/` directory, and a few utility programs including `hb-view` and `hb-shape` under the `util/` directory.

Configuration options

The instructions in the "Building HarfBuzz" section will build the source code under its default configuration. If needed, the following additional configuration options are available.

`--with-libstdc++` Allow linking with libstdc++. *(Default = no)*

> This option enables or disables linking HarfBuzz to the system's libstdc++ library.

`--with-glib` Use GLib. *(Default = auto)*

> This option enables or disables usage of the GLib library. The default setting is to check for the presence of GLib and, if it is found, build with GLib support. GLib is native to GNU/Linux systems but is available on other operating system as well.

`--with-gobject` Use GObject. *(Default = no)*

> This option enables or disables usage of the GObject library. The default setting is to check for the presence of GObject and, if it is found, build

with GObject support. GObject is native to GNU/Linux systems but is available on other operating system as well.

`--with-cairo` Use Cairo. *(Default = auto)*

This option enables or disables usage of the Cairo graphics-rendering library. The default setting is to check for the presence of Cairo and, if it is found, build with Cairo support.

Note: Cairo is used only by the HarfBuzz command-line utilities, and not by the HarfBuzz library.

`--with-fontconfig` Use Fontconfig. *(Default = auto)*

This option enables or disables usage of the Fontconfig library, which provides font-matching functions and provides access to font properties. The default setting is to check for the presence of Fontconfig and, if it is found, build with Fontconfig support.

Note: Fontconfig is used only by the HarfBuzz command-line utilities, and not by the HarfBuzz library.

`--with-icu` Use the ICU library. *(Default = auto)*

This option enables or disables usage of the *International Components for Unicode* (ICU) library, which provides access to Unicode Character Database (UCD) properties as well as normalization and conversion functions. The default setting is to check for the presence of ICU and, if it is found, build with ICU support.

`--with-graphite2` Use the Graphite2 library. *(Default = no)*

This option enables or disables usage of the Graphite2 library, which provides support for the Graphite shaping model.

`--with-freetype` Use the FreeType library. *(Default = auto)*

This option enables or disables usage of the FreeType font-rendering library. The default setting is to check for the presence of FreeType and, if it is found, build with FreeType support.

`--with-uniscribe` Use the Uniscribe library (experimental). *(Default = no)*

This option enables or disables usage of the Uniscribe font-rendering library. Uniscribe is available on Windows systems. Uniscribe support is used only for testing purposes and does not need to be enabled for HarfBuzz to run on Windows systems.

`--with-directwrite` Use the DirectWrite library (experimental). *(Default = no)*

This option enables or disables usage of the DirectWrite font-rendering library. DirectWrite is available on Windows systems. DirectWrite support is used only for testing purposes and does not need to be enabled for HarfBuzz to run on Windows systems.

`--with-coretext` Use the CoreText library. *(Default = no)*

This option enables or disables usage of the CoreText library. CoreText is available on macOS and iOS systems.

`--enable-gtk-doc` Use GTK-Doc. *(Default = no)*

This option enables the building of the documentation.

Getting started with HarfBuzz

An overview of the HarfBuzz shaping API

The core of the HarfBuzz shaping API is the function `hb_shape()`. This function takes a font, a buffer containing a string of Unicode codepoints and (optionally) a list of font features as its input. It replaces the codepoints in the buffer with the corresponding glyphs from the font, correctly ordered and positioned, and with any of the optional font features applied.

In addition to holding the pre-shaping input (the Unicode codepoints that comprise the input string) and the post-shaping output (the glyphs and positions), a HarfBuzz buffer has several properties that affect shaping. The most important are the text-flow direction (e.g., left-to-right, right-to-left, top-to-bottom, or bottom-to-top), the script tag, and the language tag.

For input string buffers, flags are available to denote when the buffer represents the beginning or end of a paragraph, to indicate whether or not to visibly render Unicode `Default          Ignorable` codepoints, and to modify the cluster-merging behavior for the buffer. For shaped output buffers, the individual X and Y offsets and `advances` (the logical dimensions) of each glyph are accessible. HarfBuzz also flags glyphs as `UNSAFE_TO_BREAK` if breaking the string at that glyph (e.g., in a line-breaking or hyphenation process) would require re-shaping the text.

HarfBuzz also provides methods to compare the contents of buffers, join buffers, normalize buffer contents, and handle invalid codepoints, as well as to determine the state of a buffer (e.g., input codepoints or output glyphs). Buffer lifecycles are managed and all buffers are reference-counted.

Although the default `hb_shape()` function is sufficient for most use cases, a variant is also provide that lets you specify which of HarfBuzz's shapers to use on a buffer.

HarfBuzz can read TrueType fonts, TrueType collections, OpenType fonts, and OpenType collections. Functions are provided to query font objects about metrics, Unicode coverage, available tables and features, and variation selectors. Individual glyphs can also be queried for metrics, variations, and glyph names. OpenType variable fonts are supported, and HarfBuzz allows you to set variation-axis coordinates on font objects.

HarfBuzz provides glue code to integrate with various other libraries, including FreeType, GObject, and CoreText. Support for integrating with Uniscribe and DirectWrite is experimental at present.

Terminology

script In text shaping, a *script* is a writing system: a set of symbols, rules, and conventions that is used to represent a language or multiple languages.

In general computing lingo, the word ”script” can also be used to mean an executable program (usually one written in a human-readable programming language). For the sake of clarity, HarfBuzz documents will always use more specific terminology when referring to this meaning, such as ”Python script” or ”shell script.” In all other instances, ”script” refers to a writing system.

For developers using HarfBuzz, it is important to note the distinction between a script and a language. Most scripts are used to write a variety of different languages, and many languages may be written in more than one script.

shaper In HarfBuzz, a *shaper* is a handler for a specific script-shaping model. HarfBuzz implements separate shapers for Indic, Arabic, Thai and Lao, Khmer, Myanmar, Tibetan, Hangul, Hebrew, the Universal Shaping Engine (USE), and a default shaper for non-complex scripts.

cluster In text shaping, a *cluster* is a sequence of codepoints that must be treated as an indivisible unit. Clusters can include code-point sequences that form a ligature or base-and-mark sequences. Tracking and preserving clusters is important when shaping operations might separate or reorder code points.

HarfBuzz provides three cluster *levels* that implement different approaches to the problem of preserving clusters during shaping operations.

grapheme In linguistics, a *grapheme* is one of the indivisible units that make up a writing system or script. Often, graphemes are individual symbols (letters, numbers, punctuation marks, logograms, etc.) but, depending on the writing system, a particular grapheme might correspond to a sequence of several Unicode code points.

In practice, HarfBuzz and other text-shaping engines are not generally concerned with graphemes. However, it is important for developers using HarfBuzz to recognize that there is a difference between graphemes and shaping clusters (see above). The two concepts may overlap frequently, but there is no guarantee that they will be identical.

syllable In linguistics, a *syllable* is an a sequence of sounds that makes up a building block of a particular language. Every language has its own set of rules describing what constitutes a valid syllable.

For text-shaping purposes, the various definitions of ”syllable” are important because script-specific shaping operations may be applied at the syllable level. For example, a reordering rule might specify that a vowel mark be reordered to the beginning of the syllable.

Syllables will consist of one or more Unicode code points. The definition of a syllable for a particular writing system might correspond to how HarfBuzz identifies clusters (see above) for the same writing system. However, it is important for developers using HarfBuzz to recognize that there is a difference between syllables and shaping clusters. The two concepts may overlap frequently, but there is no guarantee that they will be identical.

A simple shaping example

Below is the simplest HarfBuzz shaping example possible.

1. Create a buffer and put your text in it.

```
#include <hb.h>
hb_buffer_t *buf;
buf = hb_buffer_create();
hb_buffer_add_utf8(buf, text, -1, 0, -1);
```

2. Set the script, language and direction of the buffer.

```
hb_buffer_set_direction(buf, HB_DIRECTION_LTR);
hb_buffer_set_script(buf, HB_SCRIPT_LATIN);
hb_buffer_set_language(buf, hb_language_from_string("en", -1));
```

3. Create a face and a font, using FreeType for now.

```
#include <hb-ft.h>
FT_New_Face(ft_library, font_path, index, &face);
FT_Set_Char_Size(face, 0, 1000, 0, 0);
hb_font_t *font = hb_ft_font_create(face);
```

4. Shape!

```
hb_shape(font, buf, NULL, 0);
```

5. Get the glyph and position information.

```
hb_glyph_info_t *glyph_info    = hb_buffer_get_glyph_infos(buf, &glyph_count);
hb_glyph_position_t *glyph_pos = hb_buffer_get_glyph_positions(buf, &glyph_count);
```

6. Iterate over each glyph.

   ```
   for (i = 0; i < glyph_count; ++i) {
       glyphid = glyph_info[i].codepoint;
       x_offset = glyph_pos[i].x_offset / 64.0;
       y_offset = glyph_pos[i].y_offset / 64.0;
       x_advance = glyph_pos[i].x_advance / 64.0;
       y_advance = glyph_pos[i].y_advance / 64.0;
       draw_glyph(glyphid, cursor_x + x_offset, cursor_y + y_offset);
       cursor_x += x_advance;
       cursor_y += y_advance;
   }
   ```

7. Tidy up.

   ```
   hb_buffer_destroy(buf);
   hb_font_destroy(hb_ft_font);
   ```

This example shows enough to get us started using HarfBuzz. In the sections that follow, we will use the remainder of HarfBuzz's API to refine and extend the example and improve its text-shaping capabilities.

Glyph information

Names and numbers

Shaping concepts

Text shaping

Text shaping is the process of transforming a sequence of Unicode codepoints that represent individual characters (letters, diacritics, tone marks, numbers, symbols, etc.) into the orthographically and linguistically correct two-dimensional layout of glyph shapes taken from a specified font.

For some writing systems (or *scripts*) and languages, the process is simple, requiring the shaper to do little more than advance the horizontal position forward by the correct amount for each successive glyph.

But, for *complex scripts*, any combination of several shaping operations may be required, and the rules for how and when they are applied vary from script to script. HarfBuzz and other shaping engines implement these rules.

The exact rules and necessary operations for a particular script constitute a shaping *model*. OpenType specifies a set of shaping models that covers all of Unicode. Other shaping models are available, however, including Graphite and Apple Advanced Typography (AAT).

Complex scripts

In text-shaping terminology, scripts are generally classified as either *complex* or *non-complex*.

Complex scripts are those for which transforming the input sequence into the final layout requires some combination of operationsMDASHsuch as context-dependent substitutions, context-dependent mark positioning, glyph-to-glyph joining, glyph reordering, or glyph stacking.

In some complex scripts, the shaping rules require that a text run be divided into syllables before the operations can be applied. Other complex scripts may apply shaping operations over entire words or over the entire text run, with no subdivision required.

Non-complex scripts, by definition, do not require these operations. However, correctly shaping a text run in a non-complex script may still involve Unicode normalization, ligature substitutions, mark positioning, kerning, and applying other font features. The key difference is that a text run in a non-complex script can be processed sequentially and in the same order as the input sequence of Unicode codepoints, without requiring an analysis stage.

Shaping operations

Shaping a complex-script text run involves transforming the input sequence of Unicode codepoints with some combination of operations that is specified in the shaping model for the script.

The specific conditions that trigger a given operation for a text run varies from script to script, as do the order that the operations are performed in and which codepoints are affected. However, the same general set of shaping operations is common to all of the complex-script shaping models.

- A *reordering* operation moves a glyph from its original (”logical”) position in the sequence to some other (”visual”) position.

  The shaping model for a given complex script might involve more than one reordering step.

- A *joining* operation replaces a glyph with an alternate form that is designed to connect with one or more of the adjacent glyphs in the sequence.
- A contextual *substitution* operation replaces either a single glyph or a subsequence of several glyphs with an alternate glyph. This substitution is performed when the original glyph or subsequence of glyphs occurs in a

specified position with respect to the surrounding sequence. For example, one substitution might be performed only when the target glyph is the first glyph in the sequence, while another substitution is performed only when a different target glyph occurs immediately after a particular string pattern.

  The shaping model for a given complex script might involve multiple contextual-substitution operations, each applying to different target glyphs and patterns, and which are performed in separate steps.

- A contextual *positioning* operation moves the horizontal and/or vertical position of a glyph. This positioning move is performed when the glyph occurs in a specified position with respect to the surrounding sequence.

  Many contextual positioning operations are used to place *mark* glyphs (such as diacritics, vowel signs, and tone markers) with respect to *base* glyphs. However, some complex scripts may use contextual positioning operations to correctly place base glyphs as well, such as when the script uses *stacking* characters.

Unicode character categories

Shaping models are typically specified with respect to how scripts are defined in the Unicode standard.

Every codepoint in the Unicode Character Database (UCD) is assigned a *Unicode General Category* (UGC), which provides the most fundamental information about the codepoint: whether the codepoint represents a *Letter*, a *Mark*, a *Number*, *Punctuation*, a *Symbol*, a *Separator*, or something else (*Other*).

These UGC properties are ”Major” categories. Each codepoint is further assigned to a ”minor” category within its Major category, such as ”Letter, uppercase” (`Lu`) or ”Letter, modifier” (`Lm`).

Shaping models are concerned primarily with Letter and Mark codepoints. The minor categories of Mark codepoints are particularly important for shaping. Marks can be nonspacing (`Mn`), spacing combining (`Mc`), or enclosing (`Me`).

In addition to the UGC property, codepoints in the Indic and Southeast Asian scripts are also assigned *Unicode Indic Syllabic Category* (UISC) and *Unicode Indic Positional Category* (UIPC) properties that provide more detailed information needed for shaping.

The UISC property sub-categorizes Letters and Marks according to common script-shaping behaviors. For example, UISC distinguishes between consonant letters, vowel letters, and vowel marks. The UIPC property sub-categorizes Mark codepoints by the relative visual position that they occupy (above, below, right, left, or in multiple positions).

Some complex scripts require that the text run be split into syllables. What constitutes a valid syllable in these scripts is specified in regular expressions, formed from the Letter and Mark codepoints, that take the UISC and UIPC properties into account.

Text runs

Real-world text usually contains codepoints from a mixture of different Unicode scripts (including punctuation, numbers, symbols, white-space characters, and other codepoints that do not belong to any script). Real-world text may also be marked up with formatting that changes font properties (including the font, font style, and font size).

For shaping purposes, all real-world text streams must be first segmented into runs that have a uniform set of properties.

In particular, shaping models always assume that every codepoint in a text run has the same *direction*, *script* tag, and *language* tag.

OpenType shaping models

OpenType provides shaping models for the following scripts:

- The *default* shaping model handles all non-complex scripts, and may also be used as a fallback for handling unrecognized scripts.
- The *Indic* shaping model handles the Indic scripts Bengali, Devanagari, Gujarati, Gurmukhi, Kannada, Malayalam, Oriya, Tamil, Telugu, and Sinhala.

  The Indic shaping model was revised significantly in 2005. To denote the change, a new set of *script tags* was assigned for Bengali, Devanagari, Gujarati, Gurmukhi, Kannada, Malayalam, Oriya, Tamil, and Telugu. For the sake of clarity, the term ”Indic2” is sometimes used to refer to the current, revised shaping model.
- The *Arabic* shaping model supports Arabic, Mongolian, N’Ko, Syriac, and several other connected or cursive scripts.
- The *Thai/Lao* shaping model supports the Thai and Lao scripts.
- The *Khmer* shaping model supports the Khmer script.
- The *Myanmar* shaping model supports the Myanmar (or Burmese) script.
- The *Tibetan* shaping model supports the Tibetan script.
- The *Hangul* shaping model supports the Hangul script.
- The *Hebrew* shaping model supports the Hebrew script.
- The *Universal Shaping Engine* (USE) shaping model supports complex scripts not covered by one of the above, script-specific shaping models,

including Javanese, Balinese, Buginese, Batak, Chakma, Lepcha, Modi, Phags-pa, Tagalog, Siddham, Sundanese, Tai Le, Tai Tham, Tai Viet, and many others.

- Text runs that do not fall under one of the above shaping models may still require processing by a shaping engine. Of particular note is *Emoji* shaping, which may involve variation-selector sequences and glyph substitution. Emoji shaping is handled by the default shaping model.

Graphite shaping

In contrast to OpenType shaping, Graphite shaping does not specify a predefined set of shaping models or a set of supported scripts.

Instead, each Graphite font contains a complete set of rules that implement the required shaping model for the intended script. These rules include finite-state machines to match sequences of codepoints to the shaping operations to perform.

Graphite shaping can perform the same shaping operations used in OpenType shaping, as well as other functions that have not been defined for OpenType shaping.

AAT shaping

In contrast to OpenType shaping, AAT shaping does not specify a predefined set of shaping models or a set of supported scripts.

Instead, each AAT font includes a complete set of rules that implement the desired shaping model for the intended script. These rules include finite-state machines to match glyph sequences and the shaping operations to perform.

Notably, AAT shaping rules are expressed for glyphs in the font, not for Unicode codepoints. AAT shaping can perform the same shaping operations used in OpenType shaping, as well as other functions that have not been defined for OpenType shaping.

The HarfBuzz object model

An overview of data types in HarfBuzz

HarfBuzz features two kinds of data types: non-opaque, pass-by-value types and opaque, heap-allocated types. This kind of separation is common in C libraries that have to provide API/ABI compatibility (almost) indefinitely.

Value types: The non-opaque, pass-by-value types include integer types, enums, and small structs. Exposing a struct in the public API makes it impossible to expand the struct in the future. As such, exposing structs is reserved for cases where it’s extremely inefficient to do otherwise.

In HarfBuzz, several structs, like `hb_glyph_info_t` and `hb_glyph_position_t`, fall into that efficiency-sensitive category and are non-opaque.

For all non-opaque structs where future extensibility may be necessary, reserved members are included to hold space for possible future members. As such, it's important to provide `equal()`, and `hash()` methods for such structs, allowing users of the API do effectively deal with the type without having to adapt their code to future changes.

Important value types provided by HarfBuzz include the structs for working with Unicode code points, glyphs, and tags for font tables and features, as well as the enums for many Unicode and OpenType properties.

Objects in HarfBuzz

Object types: Opaque struct types are used for what HarfBuzz loosely calls "objects." This doesn't have much to do with the terminology from object-oriented programming (OOP), although some of the concepts are similar.

In HarfBuzz, all object types provide certain lifecycle-management APIs. Objects are reference-counted, and constructed with various `create()` methods, referenced via `reference()` and dereferenced using `destroy()`.

For example, the `hb_buffer_t` object has `hb_buffer_create()` as its constructor, `hb_buffer_reference()` to reference, and `hb_buffer_destroy()` to dereference.

After construction, each object's properties are accessible only through the setter and getter functions described in the API Reference manual.

Key object types provided by HarfBuzz include:

- *blobs*, which act as low-level wrappers around binary data. Blobs are typically used to hold the contents of a binary font file.
- *faces*, which represent typefaces from a font file, but without specific parameters (such as size) set.
- *fonts*, which represent instances of a face with all of their parameters specified.
- *buffers*, which hold Unicode code points for characters (before shaping) and the shaped glyph output (after shaping).
- *shape plans*, which store the settings that HarfBuzz will use when shaping a particular text segment. Shape plans are not generally used by client programs directly, but as we will see in a later chapter, they are still valuable to understand.

Object lifecycle management

Each object type in HarfBuzz provides a `create()` method. Some object types provide additional variants of `create()` to handle special cases or to speed up common tasks; those variants are documented in the API reference. For example, `hb_blob_create_from_file()` constructs a new blob directly from the contents of a file.

All objects are created with an initial reference count of `1`. Client programs can increase the reference count on an object by calling its `reference()` method. Whenever a client program is finished with an object, it should call its corresponding `destroy()` method. The destroy method will decrease the reference count on the object and, whenever the reference count reaches zero, it will also destroy the object and free all of the associated memory.

All of HarfBuzz's object-lifecycle-management APIs are thread-safe (unless you compiled HarfBuzz from source with the `HB_NO_MT` configuration flag), even when the object as a whole is not thread-safe. It is also permissible to `reference()` or to `destroy()` the `NULL` value.

Some objects are thread-safe after they have been constructed and set up. The general pattern is to `create()` the object, make a few `set_*()` calls to set up the object, and then use it without further modification.

To ensure that such an object is not modified, client programs can explicitly mark an object as immutable. HarfBuzz provides `make_immutable()` methods to mark an object as immutable and `is_immutable()` methods to test whether or not an object is immutable. Attempts to use setter functions on immutable objects will fail silently; see the API Reference manual for specifics.

Note also that there are no ”make mutable” methods. If client programs need to alter an object previously marked as immutable, they will need to make a duplicate of the original.

Finally, object constructors (and, indeed, as much of the shaping API as possible) will never return `NULL`. Instead, if there is an allocation error, each constructor will return an “empty” object singleton.

These empty-object singletons are inert and safe (although typically useless) to pass around. This design choice avoids having to check for `NULL` pointers all throughout the code.

In addition, this “empty” object singleton can also be accessed using the `get_empty()` method of the object type in question.

User data

To better integrate with client programs, HarfBuzz's objects offer a ”user data” mechanism that can be used to attach arbitrary data to the object. User-data attachment can be useful for tying the lifecycles of various pieces of data together, or for creating language bindings.

Each object type has a `set_user_data()` method and a `get_user_data()` method. The `set_user_data()` methods take a client-provided `key` and a pointer, `user_data`, pointing to the data itself. Once the key-data pair has been attached to the object, the `get_user_data()` method can be called with the key, returning the `user_data` pointer.

The `set_user_data()` methods also support an optional `destroy` callback. Client programs can set the `destroy` callback and receive notification from HarfBuzz whenever the object is destructed.

Finally, each `set_user_data()` method allows the client program to set a `replace` Boolean indicating whether or not the function call should replace any existing `user_data` associated with the specified key.

Blobs

While most of HarfBuzz's object types are specific to the shaping process, *blobs* are somewhat different.

Blobs are an abstraction desgined to negotiate lifecycle and permissions for raw pieces of data. For example, when you load the raw font data into memory and want to pass it to HarfBuzz, you do so in a `hb_blob_t` wrapper.

This allows you to take advantage of HarffBuzz's reference-counting and `destroy` callbacks. If you allocated the memory for the data using `malloc()`, you would create the blob using

```
hb_blob_create (data, length, HB_MEMORY_MODE_WRITABLE, data, free)
```

That way, HarfBuzz will call `free()` on the allocated memory whenever the blob drops its last reference and is deconstructed. Consequently, the user code can stop worrying about freeing memory and let the reference-counting machinery take care of that.

Buffers, language, script and direction

The input to the HarfBuzz shaper is a series of Unicode characters, stored in a buffer. In this chapter, we'll look at how to set up a buffer with the text that we want and how to customize the properties of the buffer. We'll also look at a piece of lower-level machinery that you will need to understand before proceeding: the functions that HarfBuzz uses to retrieve Unicode information.

After shaping is complete, HarfBuzz puts its output back into the buffer. But getting that output requires setting up a face and a font first, so we will look at that in the next chapter instead of here.

Creating and destroying buffers

As we saw in our *Getting Started* example, a buffer is created and initialized with `hb_buffer_create()`. This produces a new, empty buffer object, instantiated with some default values and ready to accept your Unicode strings.

HarfBuzz manages the memory of objects (such as buffers) that it creates, so you don't have to. When you have finished working on a buffer, you can call `hb_buffer_destroy()`:

```
hb_buffer_t *buf = hb_buffer_create();
...
hb_buffer_destroy(buf);
```

This will destroy the object and free its associated memory - unless some other part of the program holds a reference to this buffer. If you acquire a HarfBuzz buffer from another subsystem and want to ensure that it is not garbage collected by someone else destroying it, you should increase its reference count:

```
void somefunc(hb_buffer_t *buf) {
buf = hb_buffer_reference(buf);
...
```

And then decrease it once you're done with it:

```
hb_buffer_destroy(buf);
}
```

While we are on the subject of reference-counting buffers, it is worth noting that an individual buffer can only meaningfully be used by one thread at a time.

To throw away all the data in your buffer and start from scratch, call `hb_buffer_reset(buf)`. If you want to throw away the string in the buffer but keep the options, you can instead call `hb_buffer_clear_contents(buf)`.

Adding text to the buffer

Now we have a brand new HarfBuzz buffer. Let's start filling it with text! From HarfBuzz's perspective, a buffer is just a stream of Unicode code points, but your input string is probably in one of the standard Unicode character encodings (UTF-8, UTF-16, or UTF-32). HarfBuzz provides convenience functions that accept each of these encodings: `hb_buffer_add_utf8()`, `hb_buffer_add_utf16()`, and `hb_buffer_add_utf32()`. Other than the character encoding they accept, they function identically.

You can add UTF-8 text to a buffer by passing in the text array, the array's length, an offset into the array for the first character to add, and the length of the segment to add:

```
hb_buffer_add_utf8 (hb_buffer_t *buf,
                    const char *text,
                    int text_length,
                    unsigned int item_offset,
                    int item_length)
```

So, in practice, you can say:

```
hb_buffer_add_utf8(buf, text, strlen(text), 0, strlen(text));
```

This will append your new characters to `buf`, not replace its existing contents. Also, note that you can use `-1` in place of the first instance of `strlen(text)` if your text array is NULL-terminated. Similarly, you can also use `-1` as the final argument want to add its full contents.

Whatever start `item_offset` and `item_length` you provide, HarfBuzz will also attempt to grab the five characters *before* the offset point and the five characters *after* the designated end. These are the before and after "context" segments, which are used internally for HarfBuzz to make shaping decisions. They will not be part of the final output, but they ensure that HarfBuzz's script-specific shaping operations are correct. If there are fewer than five characters available for the before or after contexts, HarfBuzz will just grab what is there.

For longer text runs, such as full paragraphs, it might be tempting to only add smaller sub-segments to a buffer and shape them in piecemeal fashion. Generally, this is not a good idea, however, because a lot of shaping decisions are dependent on this context information. For example, in Arabic and other connected scripts, HarfBuzz needs to know the code points before and after each character in order to correctly determine which glyph to return.

The safest approach is to add all of the text available, then use `item_offset` and `item_length` to indicate which characters you want shaped, so that HarfBuzz has access to any context.

You can also add Unicode code points directly with `hb_buffer_add_codepoints()`. The arguments to this function are the same as those for the UTF encodings. But it is particularly important to note that HarfBuzz does not do validity checking on the text that is added to a buffer. Invalid code points will be replaced, but it is up to you to do any deep-sanity checking necessary.

Setting buffer properties

Buffers containing input characters still need several properties set before HarfBuzz can shape their text correctly.

Initially, all buffers are set to the `HB_BUFFER_CONTENT_TYPE_INVALID` content type. After adding text, the buffer should be set to `HB_BUFFER_CONTENT_TYPE_UNICODE` instead, which indicates that it contains un-shaped input characters. After shaping, the buffer will have the `HB_BUFFER_CONTENT_TYPE_GLYPHS` content type.

`hb_buffer_add_utf8()` and the other UTF functions set the content type of their buffer automatically. But if you are reusing a buffer you may want to check its state with `hb_buffer_get_content_type(buffer)`. If necessary you can set the content type with

```
hb_buffer_set_content_type(buf, HB_BUFFER_CONTENT_TYPE_UNICODE);
```

to prepare for shaping.

Buffers also need to carry information about the script, language, and text direction of their contents. You can set these properties individually:

```
hb_buffer_set_direction(buf, HB_DIRECTION_LTR);
hb_buffer_set_script(buf, HB_SCRIPT_LATIN);
hb_buffer_set_language(buf, hb_language_from_string("en", -1));
```

However, since these properties are often repeated for multiple text runs, you can also save them in a `hb_segment_properties_t` for reuse:

```
hb_segment_properties_t *savedprops;
hb_buffer_get_segment_properties (buf, savedprops);
...
hb_buffer_set_segment_properties (buf2, savedprops);
```

HarfBuzz also provides getter functions to retrieve a buffer's direction, script, and language properties individually.

HarfBuzz recognizes four text directions in `hb_direction_t`: left-to-right (`HB_DIRECTION_LTR`), right-to-left (`HB_DIRECTION_RTL`), top-to-bottom (`HB_DIRECTION_TTB`), and bottom-to-top (`HB_DIRECTION_BTT`). For the script property, HarfBuzz uses identifiers based on the ISO 15924 standard. For languages, HarfBuzz uses tags based on the IETF BCP 47 standard.

Helper functions are provided to convert character strings into the necessary script and language tag types.

Two additional buffer properties to be aware of are the "invisible glyph" and the replacement code point. The replacement code point is inserted into buffer output in place of any invalid code points encountered in the input. By default, it is the Unicode `REPLACEMENT CHARACTER` code point, `U+FFFD` " ". You can change this with

```
hb_buffer_set_replacement_codepoint(buf, replacement);
```

passing in the replacement Unicode code point as the `replacement` parameter.

The invisible glyph is used to replace all output glyphs that are invisible. By default, the standard space character `U+0020` is used; you can replace this (for example, when using a font that provides script-specific spaces) with

```
hb_buffer_set_invisible_glyph(buf, replacement_glyph);
```

Do note that in the `replacement_glyph` parameter, you must provide the glyph ID of the replacement you wish to use, not the Unicode code point.

HarfBuzz supports a few additional flags you might want to set on your buffer under certain circumstances. The `HB_BUFFER_FLAG_BOT` and `HB_BUFFER_FLAG_EOT` flags tell HarfBuzz that the buffer represents the beginning or end (respectively) of a text element (such as a paragraph or other block). Knowing this allows HarfBuzz to apply certain contextual font features when shaping, such as initial or final variants in connected scripts.

`HB_BUFFER_FLAG_PRESERVE_DEFAULT_IGNORABLES` tells HarfBuzz not to hide glyphs with the `Default_Ignorable` property in Unicode. This property designates control characters and other non-printing code points, such as joiners and variation selectors. Normally HarfBuzz replaces them in the output buffer with zero-width space glyphs (using the ”invisible glyph” property discussed above); setting this flag causes them to be printed, which can be helpful for troubleshooting.

Conversely, setting the `HB_BUFFER_FLAG_REMOVE_DEFAULT_IGNORABLES` flag tells HarfBuzz to remove `Default_Ignorable` glyphs from the output buffer entirely. Finally, setting the `HB_BUFFER_FLAG_DO_NOT_INSERT_DOTTED_CIRCLE` flag tells HarfBuzz not to insert the dotted-circle glyph (`U+25CC`, ” ”), which is normally inserted into buffer output when broken character sequences are encountered (such as combining marks that are not attached to a base character).

Customizing Unicode functions

HarfBuzz requires some simple functions for accessing information from the Unicode Character Database (such as the `General_Category` (gc) and `Script` (sc) properties) that is useful for shaping, as well as some useful operations like composing and decomposing code points.

HarfBuzz includes its own internal, lightweight set of Unicode functions. At build time, it is also possible to compile support for some other options, such as the Unicode functions provided by GLib or the International Components for Unicode (ICU) library. Generally, this option is only of interest for client programs that have specific integration requirements or that do a significant amount of customization.

If your program has access to other Unicode functions, however, such as through a system library or application framework, you might prefer to use those instead of the built-in options. HarfBuzz supports this by implementing its Unicode functions as a set of virtual methods that you can replace — without otherwise affecting HarfBuzz's functionality.

The Unicode functions are specified in a structure called `unicode_funcs` which is attached to each buffer. But even though `unicode_funcs` is associated with a `hb_buffer_t`, the functions themselves are called by other HarfBuzz APIs that access buffers, so it would be unwise for you to hook different functions into different buffers.

In addition, you can mark your `unicode_funcs` as immutable by calling `hb_unicode_funcs_make_immutable (ufuncs)`. This is especially useful if your code is a library or framework that will have its own client programs. By marking your Unicode function choices as immutable, you prevent your own client programs from changing the `unicode_funcs` configuration and introducing inconsistencies and errors downstream.

You can retrieve the Unicode-functions configuration for your buffer by calling `hb_buffer_get_unicode_funcs()`:

```
hb_unicode_funcs_t *ufunctions;
ufunctions = hb_buffer_get_unicode_funcs(buf);
```

The current version of `unicode_funcs` uses six functions:

- `hb_unicode_combining_class_func_t`: returns the Canonical Combining Class of a code point.
- `hb_unicode_general_category_func_t`: returns the General Category (gc) of a code point.
- `hb_unicode_mirroring_func_t`: returns the Mirroring Glyph code point (for bi-directional replacement) of a code point.
- `hb_unicode_script_func_t`: returns the Script (sc) property of a code point.
- `hb_unicode_compose_func_t`: returns the canonical composition of a sequence of two code points.
- `hb_unicode_decompose_func_t`: returns the canonical decomposition of a code point.

Note, however, that future HarfBuzz releases may alter this set.

Each Unicode function has a corresponding setter, with which you can assign a callback to your replacement function. For example, to replace `hb_unicode_general_category_func_t`, you can call

```
hb_unicode_funcs_set_general_category_func (*ufuncs, func, *user_data, destroy)
```

Virtualizing this set of Unicode functions is primarily intended to improve portability. There is no need for every client program to make the effort to replace the default options, so if you are unsure, do not feel any pressure to customize `unicode_funcs`.

Fonts, faces, and output

In the previous chapter, we saw how to set up a buffer and fill it with text as Unicode code points. In order to shape this buffer text with HarfBuzz, you will need also need a font object.

HarfBuzz provides abstractions to help you cache and reuse the heavier parts of working with binary fonts, so we will look at how to do that. We will also look at how to work with the FreeType font-rendering library and at how you can customize HarfBuzz to work with other libraries.

Finally, we will look at how to work with OpenType variable fonts, the latest update to the OpenType font format, and at some other recent additions to OpenType.

Font and face objects

The outcome of shaping a run of text depends on the contents of a specific font file (such as the substitutions and positioning moves in the 'GSUB' and 'GPOS' tables), so HarfBuzz makes accessing those internals fast.

An `hb_face_t` represents a *face* in HarfBuzz. This data type is a wrapper around an `hb_blob_t` blob that holds the contents of a binary font file. Since HarfBuzz supports TrueType Collections and OpenType Collections (each of which can include multiple typefaces), a HarfBuzz face also requires an index number specifying which typeface in the file you want to use. Most of the font files you will encounter in the wild include just a single face, however, so most of the time you would pass in `0` as the index when you create a face:

```
hb_blob_t* blob = hb_blob_create_from_file(file);
...
hb_face_t* face = hb_face_create(blob, 0);
```

On its own, a face object is not quite ready to use for shaping. The typeface must be set to a specific point size in order for some details (such as hinting) to work. In addition, if the font file in question is an OpenType Variable Font, then you may need to specify one or variation-axis settings (or a named instance) in order to get the output you need.

In HarfBuzz, you do this by creating a *font* object from your face.

Font objects also have the advantage of being considerably lighter-weight than face objects (remember that a face contains the contents of a binary font file mapped into memory). As a result, you can cache and reuse a font object, but you could also create a new one for each additional size you needed. Creating new fonts incurs some additional overhead, of course, but whether or not it is excessive is your call in the end. In contrast, face objects are substantially larger, and you really should cache them and reuse them whenever possible.

You can create a font object from a face object:

```
hb_font_t* hb_font = hb_font_create(hb_face);
```

After creating a font, there are a few properties you should set. Many fonts enable and disable hints based on the size it is used at, so setting this is important for font objects. `hb_font_set_ppem(font, x_ppem,      y_ppem)` sets the pixels-per-EM value of the font. You can also set the point size of the font with `hb_font_set_ptem(font, ptem)`. HarfBuzz uses the industry standard 72 points per inch.

HarfBuzz lets you specify the degree subpixel precision you want through a scaling factor. You can set horizontal and vertical scaling factors on the font by calling `hb_font_set_scale(font, x_scale,      y_scale)`.

There may be times when you are handed a font object and need to access the face object that it comes from. For that, you can call

```
hb_face = hb_font_get_face(hb_font);
```

You can also create a font object from an existing font object using the `hb_font_create_sub_font()` function. This creates a child font object that is initiated with the same attributes as its parent; it can be used to quickly set up a new font for the purpose of overriding a specific font-functions method.

All face objects and font objects are lifecycle-managed by HarfBuzz. After creating a face, you increase its reference count with `hb_face_reference(face)` and decrease it with `hb_face_destroy(face)`. Likewise, you increase the reference count on a font with `hb_font_reference(font)` and decrease it with `hb_font_destroy(font)`.

You can also attach user data to face objects and font objects.

Customizing font functions

During shaping, HarfBuzz frequently needs to query font objects to get at the contents and parameters of the glyphs in a font file. It includes a built-in set of functions that is tailored to working with OpenType fonts. However, as was the case with Unicode functions in the buffers chapter, HarfBuzz also wants to make it easy for you to assign a substitute set of font functions if you are developing a program to work with a library or platform that provides its own font functions.

Therefore, the HarfBuzz API defines a set of virtual methods for accessing font-object properties, and you can replace the defaults with your own selections without interfering with the shaping process. Each font object in HarfBuzz includes a structure called `font_funcs` that serves as a vtable for the font object. The virtual methods in `font_funcs` are:

- `hb_font_get_font_h_extents_func_t`: returns the extents of the font for horizontal text.
- `hb_font_get_font_v_extents_func_t`: returns the extents of the font for vertical text.
- `hb_font_get_nominal_glyph_func_t`: returns the font's nominal glyph for a given code point.
- `hb_font_get_variation_glyph_func_t`: returns the font's glyph for a given code point when it is followed by a given Variation Selector.
- `hb_font_get_nominal_glyphs_func_t`: returns the font's nominal glyphs for a series of code points.
- `hb_font_get_glyph_advance_func_t`: returns the advance for a glyph.
- `hb_font_get_glyph_h_advance_func_t`: returns the advance for a glyph for horizontal text.
- `hb_font_get_glyph_v_advance_func_t`:returns the advance for a glyph for vertical text.
- `hb_font_get_glyph_advances_func_t`: returns the advances for a series of glyphs.
- `hb_font_get_glyph_h_advances_func_t`: returns the advances for a series of glyphs for horizontal text .
- `hb_font_get_glyph_v_advances_func_t`: returns the advances for a series of glyphs for vertical text.
- `hb_font_get_glyph_origin_func_t`: returns the origin coordinates of a glyph.
- `hb_font_get_glyph_h_origin_func_t`: returns the origin coordinates of a glyph for horizontal text.
- `hb_font_get_glyph_v_origin_func_t`: returns the origin coordinates of a glyph for vertical text.
- `hb_font_get_glyph_extents_func_t`: returns the extents for a glyph.
- `hb_font_get_glyph_contour_point_func_t`: returns the coordinates of a specific contour point from a glyph.
- `hb_font_get_glyph_name_func_t`: returns the name of a glyph (from its glyph index).

- `hb_font_get_glyph_from_name_func_t`: returns the glyph index that corresponds to a given glyph name.

You can fetch the font-functions configuration for a font object by calling `hb_font_get_font_funcs()`:

```
hb_font_funcs_t *ffunctions;
ffunctions = hb_font_get_font_funcs (font);
```

The individual methods can each be replaced with their own setter function, such as `hb_font_funcs_set_nominal_glyph_func(*ffunctions, func, *user_data, destroy)`.

Font-functions structures can be reused for multiple font objects, and can be reference counted with `hb_font_funcs_reference()` and `hb_font_funcs_destroy()`. Just like other objects in HarfBuzz, you can set user-data for each font-functions structure and assign a destroy callback for it.

You can also mark a font-functions structure as immutable, with `hb_font_funcs_make_immutable()`. This is especially useful if your code is a library or framework that will have its own client programs. By marking your font-functions structures as immutable, you prevent your client programs from changing the configuration and introducing inconsistencies and errors downstream.

Font objects and HarfBuzz's native OpenType implementation

By default, whenever HarfBuzz creates a font object, it will configure the font to use a built-in set of font functions that supports contemporary OpenType font internals. If you want to work with OpenType or TrueType fonts, you should be able to use these functions without difficulty.

Many of the methods in the font-functions structure deal with the fundamental properties of glyphs that are required for shaping text: extents (the maximums and minimums on each axis), origins (the `(0,0)` coordinate point which glyphs are drawn in reference to), and advances (the amount that the cursor needs to be moved after drawing each glyph, including any empty space for the glyph's side bearings).

As you can see in the list of functions, there are separate "horizontal" and "vertical" variants depending on whether the text is set in the horizontal or vertical direction. For some scripts, fonts that are designed to support text set horizontally or vertically (for example, in Japanese) may include metrics for both text directions. When fonts don't include this information, HarfBuzz does its best to transform what the font provides.

In addition to the direction-specific functions, HarfBuzz provides some higher-level functions for fetching information like extents and advances for a glyph. If you call

```
hb_font_get_glyph_advance_for_direction(font, direction, extents);
```

then you can provide any `hb_direction_t` as the `direction` parameter, and HarfBuzz will use the correct function variant for the text direction. There are similar higher-level versions of the functions for fetching extents, origin coordinates, and contour-point coordinates. There are also addition and subtraction functions for moving points with respect to the origin.

There are also methods for fetching the glyph ID that corresponds to a Unicode code point (possibly when followed by a variation-selector code point), fetching the glyph name from the font, and fetching the glyph ID that corresponds to a glyph name you already have.

HarfBuzz also provides functions for converting between glyph names and string variables. `hb_font_glyph_to_string(font, glyph, s,        size)` retrieves the name for the glyph ID `glyph` from the font object. It generates a generic name of the form `gidDDD` (where DDD is the glyph index) if there is no name for the glyph in the font. The `hb_font_glyph_from_string(font, s, len,        glyph)` takes an input string `s` and looks for a glyph with that name in the font, returning its glyph ID in the `glyph` output parameter. It automatically parses `gidDDD` and `uniUUUU` strings.

Working with OpenType Variable Fonts

If you are working with OpenType Variable Fonts, there are a few additional functions you should use to specify the variation-axis settings of your font object. Without doing so, your variable font's font object can still be used, but only at the default setting for every axis (which, of course, is sometimes what you want, but does not cover general usage).

HarfBuzz manages variation settings in the `hb_variation_t` data type, which holds a tag for the variation-axis identifier tag and a value for its setting. You can retrieve the list of variation axes in a font binary from the face object (not from a font object, notably) by calling `hb_ot_var_get_axis_count(face)` to find the number of axes, then using `hb_ot_var_get_axis_infos()` to collect the axis structures:

```
axes = hb_ot_var_get_axis_count(face);
...
hb_ot_var_get_axis_infos(face, 0, axes, axes_array);
```

For each axis returned in the array, you can can access the identifier in its tag. HarfBuzz also has tag definitions predefined for the five standard axes specified in OpenType (`ital` for italic, `opsz` for optical size, `slnt` for slant, `wdth` for width, and `wght` for weight). Each axis also has a min_value, a default_value, and a max_value.

To set your font object's variation settings, you call the `hb_font_set_variations()` function with an array of `hb_variation_t` variation settings. Let's say our font has weight and width axes. We need to specify each of the axes by tag and assign a value on the axis:

```
unsigned int variation_count = 2;
hb_variation_t variation_data[variation_count];
variation_data[0].tag = HB_OT_TAG_VAR_AXIS_WIDTH;
variation_data[1].tag = HB_OT_TAG_VAR_AXIS_WEIGHT;
variation_data[0].value = 80;
variation_data[1].value = 750;
...
hb_font_set_variations(font, variation_data, variation_count);
```

That should give us a slightly condensed font ("normal" on the `wdth` axis is 100) at a noticeably bolder weight ("regular" is 400 on the `wght` axis).

In practice, though, you should always check that the value you want to set on the axis is within the [min_value,max_value] range actually implemented in the font's variation axis. After all, a font might only provide lighter-than-regular weights, and setting a heavier value on the `wght` axis will not change that.

Once your variation settings are specified on your font object, however, shaping with a variable font is just like shaping a static font.

Shaping and shape plans

Once you have your face and font objects configured as desired and your input buffer is filled with the characters you need to shape, all you need to do is call `hb_shape()`.

HarfBuzz will return the shaped version of the text in the same buffer that you provided, but it will be in output mode. At that point, you can iterate through the glyphs in the buffer, drawing each one at the specified position or handing them off to the appropriate graphics library.

For the most part, HarfBuzz's shaping step is straightforward from the outside. But that doesn't mean there will never be cases where you want to look under the hood and see what is happening on the inside. HarfBuzz provides facilities for doing that, too.

Shaping and buffer output

The `hb_shape()` function call takes four arguments: the font object to use, the buffer of characters to shape, an array of user-specified features to apply, and the length of that feature array. The feature array can be NULL, so for the sake of simplicity we will start with that case.

Internally, HarfBuzz looks at the tables of the font file to determine where glyph classes, substitutions, and positioning are defined, using that information to decide which *shaper* to use (`ot` for OpenType fonts, `aat` for Apple Advanced Typography fonts, and so on). It also looks at the direction, script, and language properties of the segment to figure out which script-specific shaping model is needed (at least, in shapers that support multiple options).

If a font has a GDEF table, then that is used for glyph classes; if not, HarfBuzz will fall back to Unicode categorization by code point. If a font has an AAT "morx" table, then it is used for substitutions; if not, but there is a GSUB table, then the GSUB table is used. If the font has an AAT "kerx" table, then it is used for positioning; if not, but there is a GPOS table, then the GPOS table is used. If neither table is found, but there is a "kern" table, then HarfBuzz will use the "kern" table. If there is no "kerx", no GPOS, and no "kern", HarfBuzz will fall back to positioning marks itself.

With a well-behaved OpenType font, you expect GDEF, GSUB, and GPOS tables to all be applied. HarfBuzz implements the script-specific shaping models in internal functions, rather than in the public API.

The algorithms used for complex scripts can be quite involved; HarfBuzz tries to be compatible with the OpenType Layout specification and, wherever there is any ambiguity, HarfBuzz attempts to replicate the output of Microsoft's Uniscribe engine. See the Microsoft Typography pages for more detail.

In general, though, all that you need to know is that `hb_shape()` returns the results of shaping in the same buffer that you provided. The buffer's content type will now be set to `HB_BUFFER_CONTENT_TYPE_GLYPHS`, indicating that it contains shaped output, rather than input text. You can now extract the glyph information and positioning arrays:

```
hb_glyph_info_t *glyph_info    = hb_buffer_get_glyph_infos(buf, &glyph_count);
hb_glyph_position_t *glyph_pos = hb_buffer_get_glyph_positions(buf, &glyph_count);
```

The glyph information array holds a `hb_glyph_info_t` for each output glyph, which has two fields: `codepoint` and `cluster`. Whereas, in the input buffer, the `codepoint` field contained the Unicode code point, it now contains the glyph ID of the corresponding glyph in the font. The `cluster` field is an integer that you can use to help identify when shaping has reordered, split, or combined code points; we will say more about that in the next chapter.

The glyph positions array holds a corresponding `hb_glyph_position_t` for each output glyph, containing four fields: `x_advance`, `y_advance`, `x_offset`, and `y_offset`. The advances tell you how far you need to move the drawing point after drawing this glyph, depending on whether you are setting horizontal text (in which case you will have x advances) or vertical text (for which you will have y advances). The x and y offsets tell you where to move to start drawing the glyph; usually you will have both and x and a y offset, regardless of the text direction.

Most of the time, you will rely on a font-rendering library or other graphics library to do the actual drawing of glyphs, so you will need to iterate through the glyphs in the buffer and pass the corresponding values off.

OpenType features

OpenType features enable fonts to include smart behavior, implemented as ”lookup” rules stored in the GSUB and GPOS tables. The OpenType specification defines a long list of standard features that fonts can use for these behaviors; each feature has a four-character reserved name and a well-defined semantic meaning.

Some OpenType features are defined for the purpose of supporting complex-script shaping, and are automatically activated, but only when a buffer's script property is set to a script that the feature supports.

Other features are more generic and can apply to several (or any) script, and shaping engines are expected to implement them. By default, HarfBuzz activates several of these features on every text run. They include `abvm`, `blwm`, `ccmp`, `locl`, `mark`, `mkmk`, and `rlig`.

In addition, if the text direction is horizontal, HarfBuzz also applies the `calt`, `clig`, `curs`, `dist`, `kern`, `liga`, `rclt`, and `frac` features.

If the text direction is vertical, HarfBuzz applies the `vert` feature by default.

Still other features are designed to be purely optional and left up to the application or the end user to enable or disable as desired.

You can adjust the set of features that HarfBuzz applies to a buffer by supplying an array of `hb_feature_t` features as the third argument to `hb_shape()`. For a simple case, let's just enable the `dlig` feature, which turns on any ”discretionary” ligatures in the font:

```
hb_feature_t userfeatures[1];
userfeatures[0].tag = HB_TAG('d','l','i','g');
userfeatures[0].value = 1;
userfeatures[0].start = HB_FEATURE_GLOBAL_START;
userfeatures[0].end = HB_FEATURE_GLOBAL_END;
```

`HB_FEATURE_GLOBAL_END` and `HB_FEATURE_GLOBAL_END` are macros we can use to indicate that the features will be applied to the entire buffer. We could also have used a literal `0` for the start and a `-1` to indicate the end of the buffer (or have selected other start and end positions, if needed).

When we pass the `userfeatures` array to `hb_shape()`, any discretionary ligature substitutions from our font that match the text in our buffer will get performed:

```
hb_shape(font, buf, userfeatures, num_features);
```

Just like we enabled the `dlig` feature by setting its `value` to `1`, you would disable a feature by setting its `value` to `0`. Some features can take other `value` settings; be sure you read the full specification of each feature tag to understand what it does and how to control it.

Shaper selection

The basic version of `hb_shape()` determines its shaping strategy based on examining the capabilities of the font file. OpenType font tables cause HarfBuzz to try the `ot` shaper, while AAT font tables cause HarfBuzz to try the `aat` shaper.

In the real world, however, a font might include some unusual mix of tables, or one of the tables might simply be broken for the script you need to shape. So, sometimes, you might not want to rely on HarfBuzz's process for deciding what to do, and just tell `hb_shape()` what you want it to try.

`hb_shape_full()` is an alternate shaping function that lets you supply a list of shapers for HarfBuzz to try, in order, when shaping your buffer. For example, if you have determined that HarfBuzz's attempts to work around broken tables gives you better results than the AAT shaper itself does, you might move the AAT shaper to the end of your list of preferences and call `hb_shape_full()`

```
char *shaperprefs[3] = {"ot", "default", "aat"};
...
hb_shape_full(font, buf, userfeatures, num_features, shaperprefs);
```

to get results you are happier with.

You may also want to call `hb_shape_list_shapers()` to get a list of the shapers that were built at compile time in your copy of HarfBuzz.

Plans and caching

Internally, HarfBuzz uses a structure called a shape plan to track its decisions about how to shape the contents of a buffer. The `hb_shape()` function builds up the shape plan by examining segment properties and by inspecting the contents of the font.

This process can involve some decision-making and trade-offs — for example, HarfBuzz inspects the GSUB and GPOS lookups for the script and language tags set on the segment properties, but it falls back on the lookups under the `DFLT` tag (and sometimes other common tags) if there are actually no lookups for the tag requested.

HarfBuzz also includes some work-arounds for handling well-known older font conventions that do not follow OpenType or Unicode specifications, for buggy system fonts, and for peculiarities of Microsoft Uniscribe. All of that means that a shape plan, while not something that you should edit directly in client code, still might be an object that you want to inspect. Furthermore, if resources are tight, you might want to cache the shape plan that HarfBuzz builds for your buffer and font, so that you do not have to rebuild it for every shaping call.

You can create a cacheable shape plan with `hb_shape_plan_create_cached(face, props, user_features, num_user_features, shaper_list)`, where `face` is a face object (not a font object, notably), `props` is an `hb_segment_properties_t`, `user_features` is an array of `hb_feature_t`s (with length `num_user_features`), and `shaper_list` is a list of shapers to try.

Shape plans are objects in HarfBuzz, so there are reference-counting functions and user-data attachment functions you can use. `hb_shape_plan_reference(shape_plan)` increases the reference count on a shape plan, while `hb_shape_plan_destroy(shape_plan)` decreases the reference count, destroying the shape plan when the last reference is dropped.

You can attach user data to a shaper (with a key) using the `hb_shape_plan_set_user_data(shape_plan,key,d` function, optionally supplying a `destroy` callback to use. You can then fetch the user data attached to a shape plan with `hb_shape_plan_get_user_data(shape_plan, key)`.

Clusters

Clusters and shaping

In text shaping, a *cluster* is a sequence of characters that needs to be treated as a single, indivisible unit. A single letter or symbol can be a cluster of its own. Other clusters correspond to longer subsequences of the input code points MDASH such as a ligature or conjunct form MDASH and require the shaper to ensure that the cluster is not broken during the shaping process.

A cluster is distinct from a *grapheme*, which is the smallest unit of meaning in a writing system or script.

The definitions of the two terms are similar. However, clusters are only relevant for script shaping and glyph layout. In contrast, graphemes are a property of the underlying script, and are of interest when client programs implement orthographic or linguistic functionality.

For example, two individual letters are often two separate graphemes. When two letters form a ligature, however, they combine into a single glyph. They are then part of the same cluster and are treated as a unit by the shaping engine MDASH even though the two original, underlying letters remain separate graphemes.

HarfBuzz is concerned with clusters, *not* with graphemes MDASH although client programs using HarfBuzz may still care about graphemes for other reasons from time to time.

During the shaping process, there are several shaping operations that may merge adjacent characters (for example, when two code points form a ligature or a conjunct form and are replaced by a single glyph) or split one character into several (for example, when decomposing a code point through the `ccmp` feature). Operations like these alter clusters; HarfBuzz tracks the changes to ensure that no clusters get lost or broken during shaping.

HarfBuzz records cluster information independently from how shaping operations affect the individual glyphs returned in an output buffer. Consequently, a client program using HarfBuzz can utilize the cluster information to implement features such as:

- Correctly positioning the cursor within a shaped text run, even when characters have formed ligatures, composed or decomposed, reordered, or undergone other shaping operations.
- Correctly highlighting a text selection that includes some, but not all, of the characters in a word.
- Applying text attributes (such as color or underlining) to part, but not all, of a word.
- Generating output document formats (such as PDF) with embedded text that can be fully extracted.
- Determining the mapping between input characters and output glyphs, such as which glyphs are ligatures.
- Performing line-breaking, justification, and other line-level or paragraph-level operations that must be done after shaping is complete, but which require examining character-level properties.

Working with HarfBuzz clusters

When you add text to a HarfBuzz buffer, each code point must be assigned a *cluster value.*

This cluster value is an arbitrary number; HarfBuzz uses it only to distinguish between clusters. Many client programs will use the index of each code point in the input text stream as the cluster value. This is for the sake of convenience; the actual value does not matter.

Some of the shaping operations performed by HarfBuzz MDASH such as reordering, composition, decomposition, and substitution MDASH may alter the cluster values of some characters. The final cluster values in the buffer at the end of the shaping process will indicate to client programs which subsequences of glyphs represent a cluster and, therefore, must not be separated.

In addition, client programs can query the final cluster values to discern other potentially important information about the glyphs in the output buffer (such as whether or not a ligature was formed).

For example, if the initial sequence of cluster values was:

```
0,1,2,3,4
```

and the final sequence of cluster values is:

```
0,0,3,3
```

then there are two clusters in the output buffer: the first cluster includes the first two glyphs, and the second cluster includes the third and fourth glyphs. It is also evident that a ligature or conjunct has been formed, because there are fewer glyphs in the output buffer (four) than there were code points in the input buffer (five).

Although client programs using HarfBuzz are free to assign initial cluster values in any manner they choose to, HarfBuzz does offer some useful guarantees if the cluster values are assigned in a monotonic (either non-decreasing or non-increasing) order.

For buffers in the left-to-right (LTR) or top-to-bottom (TTB) text flow direction, HarfBuzz will preserve the monotonic property: client programs are guaranteed that monotonically increasing initial cluster values will be returned as monotonically increasing final cluster values.

For buffers in the right-to-left (RTL) or bottom-to-top (BTT) text flow direction, the directionality of the buffer itself is reversed for final output as a matter of design. Therefore, HarfBuzz inverts the monotonic property: client programs are guaranteed that monotonically increasing initial cluster values will be returned as monotonically *decreasing* final cluster values.

Client programs can adjust how HarfBuzz handles clusters during shaping by setting the `cluster_level` of the buffer. HarfBuzz offers three *levels* of clustering support for this property:

- *Level 0* is the default and reproduces the behavior of the old HarfBuzz library.

  The distinguishing feature of level 0 behavior is that, at the beginning of processing the buffer, all code points that are categorized as *marks*, *modifier symbols*, or *Emoji extended pictographic* modifiers, as well as the *Zero Width Joiner* and *Zero Width Non-Joiner* code points, are assigned the cluster value of the closest preceding code point from *different* category.

  In essence, whenever a base character is followed by a mark character or a sequence of mark characters, those marks are reassigned to the same initial cluster value as the base character. This reassignment is referred to as "merging" the affected clusters. This behavior is based on the Grapheme Cluster Boundary specification in Unicode Technical Report 29.

Client programs can specify level 0 behavior for a buffer by setting its `cluster_level` to `HB_BUFFER_CLUSTER_LEVEL_MONOTONE_GRAPHEMES`.

- *Level 1* tweaks the old behavior slightly to produce better results. Therefore, level 1 clustering is recommended for code that is not required to implement backward compatibility with the old HarfBuzz.

  Level 1 differs from level 0 by not merging the clusters of marks and other modifier code points with the preceding ”base” code point’s cluster. By preserving the separate cluster values of these marks and modifier code points, script shapers can perform additional operations that might lead to improved results (for example, reordering a sequence of marks).

  Client programs can specify level 1 behavior for a buffer by setting its `cluster_level` to `HB_BUFFER_CLUSTER_LEVEL_MONOTONE_CHARACTERS`.

- *Level 2* differs significantly in how it treats cluster values. In level 2, HarfBuzz never merges clusters.

  This difference can be seen most clearly when HarfBuzz processes ligature substitutions and glyph decompositions. In level 0 and level 1, ligatures and glyph decomposition both involve merging clusters; in level 2, neither of these operations triggers a merge.

  Client programs can specify level 2 behavior for a buffer by setting its `cluster_level` to `HB_BUFFER_CLUSTER_LEVEL_CHARACTERS`.

As mentioned earlier, client programs using HarfBuzz often assign initial cluster values in a buffer by reusing the indices of the code points in the input text. This gives a sequence of cluster values that is monotonically increasing (for example, 0,1,2,3,4).

It is not *required* that the cluster values in a buffer be monotonically increasing. However, if the initial cluster values in a buffer are monotonic and the buffer is configured to use cluster level 0 or 1, then HarfBuzz guarantees that the final cluster values in the shaped buffer will also be monotonic. No such guarantee is made for cluster level 2.

In levels 0 and 1, HarfBuzz implements the following conceptual model for cluster values:

- If the sequence of input cluster values is monotonic, the sequence of cluster values will remain monotonic.
- Each cluster value represents a single cluster.
- Each cluster contains one or more glyphs and one or more characters.

In practice, this model offers several benefits. Assuming that the initial cluster values were monotonically increasing and distinct before shaping began, then, in the final output:

- All adjacent glyphs having the same final cluster value belong to the same cluster.
- Each character belongs to the cluster that has the highest cluster value *not larger than* its initial cluster value.

A clustering example for levels 0 and 1

The basic shaping operations affect clusters in a predictable manner when using level 0 or level 1:

- When two or more clusters *merge*, the resulting merged cluster takes as its cluster value the *minimum* of the incoming cluster values.
- When a cluster *decomposes*, all of the resulting child clusters inherit as their cluster value the cluster value of the parent cluster.
- When a character is *reordered*, the reordered character and all clusters that the character moves past as part of the reordering are merged into one cluster.

The functionality, guarantees, and benefits of level 0 and level 1 behavior can be seen with some examples. First, let us examine what happens with cluster values when shaping involves cluster merging with ligatures and decomposition.

Let's say we start with the following character sequence (top row) and initial cluster values (bottom row):

```
A,B,C,D,E
0,1,2,3,4
```

During shaping, HarfBuzz maps these characters to glyphs from the font. For simplicity, let us assume that each character maps to the corresponding, identical-looking glyph:

```
A,B,C,D,E
0,1,2,3,4
```

Now if, for example, `B` and `C` form a ligature, then the clusters to which they belong "merge". This merged cluster takes for its cluster value the minimum of all the cluster values of the clusters that went in to the ligature. In this case, we get:

```
A,BC,D,E
0,1 ,3,4
```

because 1 is the minimum of the set {1,2}, which were the cluster values of `B` and `C`.

Next, let us say that the `BC` ligature glyph decomposes into three components, and `D` also decomposes into two components. Whenever a cluster decomposes, its components each inherit the cluster value of their parent:

```
A,BC0,BC1,BC2,D0,D1,E
0,1  ,1  ,1  ,3 ,3 ,4
```

Next, if `BC2` and `D0` form a ligature, then their clusters (cluster values 1 and 3) merge into `min(1,3) = 1`:

```
A,BC0,BC1,BC2D0,D1,E
0,1  ,1  ,1    ,1 ,4
```

Note that the entirety of cluster 3 merges into cluster 1, not just the `D0` glyph. This reflects the fact that the cluster *must* be treated as an indivisible unit.

At this point, cluster 1 means: the character sequence `BCD` is represented by glyphs `BC0,BC1,BC2D0,D1` and cannot be broken down any further.

Reordering in levels 0 and 1

Another common operation in the more complex shapers is glyph reordering. In order to maintain a monotonic cluster sequence when glyph reordering takes place, HarfBuzz merges the clusters of everything in the reordering sequence.

For example, let us again start with the character sequence (top row) and initial cluster values (bottom row):

```
A,B,C,D,E
0,1,2,3,4
```

If `D` is reordered to the position immediately before `B`, then HarfBuzz merges the `B`, `C`, and `D` clusters MDASH all the clusters between the final position of the reordered glyph and its original position. This means that we get:

```
A,D,B,C,E
0,1,1,1,4
```

as the final cluster sequence.

Merging this many clusters is not ideal, but it is the only sensible way for HarfBuzz to maintain the guarantee that the sequence of cluster values remains monotonic and to retain the true relationship between glyphs and characters.

The distinction between levels 0 and 1

The preceding examples demonstrate the main effects of using cluster levels 0 and 1. The only difference between the two levels is this: in level 0, at the very beginning of the shaping process, HarfBuzz merges the cluster of each base character with the clusters of all Unicode marks (combining or not) and modifiers that follow it.

For example, let us start with the following character sequence (top row) and accompanying initial cluster values (bottom row):

```
A,acute,B
0,1    ,2
```

The `acute` is a Unicode mark. If HarfBuzz is using cluster level 0 on this sequence, then the `A` and `acute` clusters will merge, and the result will become:

```
A,acute,B
0,0    ,2
```

This merger is performed before any other script-shaping steps.

This initial cluster merging is the default behavior of the Windows shaping engine, and the old HarfBuzz codebase copied that behavior to maintain compatibility. Consequently, it has remained the default behavior in the new HarfBuzz codebase.

But this initial cluster-merging behavior makes it impossible for client programs to implement some features (such as to color diacritic marks differently from their base characters). That is why, in level 1, HarfBuzz does not perform the initial merging step.

For client programs that rely on HarfBuzz cluster values to perform cursor positioning, level 0 is more convenient. But relying on cluster boundaries for cursor positioning is wrong: cursor positions should be determined based on Unicode grapheme boundaries, not on shaping-cluster boundaries. As such, using level 1 clustering behavior is recommended.

One final facet of levels 0 and 1 is worth noting. HarfBuzz currently does not allow any *multiple-substitution* GSUB lookups to replace a glyph with zero glyphs (in other words, to delete a glyph).

But, in some other situations, glyphs can be deleted. In those cases, if the glyph being deleted is the last glyph of its cluster, HarfBuzz makes sure to merge the deleted glyph's cluster with a neighboring cluster.

This is done primarily to make sure that the starting cluster of the text always has the cluster index pointing to the start of the text for the run; more than one client program currently relies on this guarantee.

Incidentally, Apple's CoreText does something different to maintain the same promise: it inserts a glyph with id 65535 at the beginning of the glyph string if the glyph corresponding to the first character in the run was deleted. HarfBuzz might do something similar in the future.

Level 2

HarfBuzz's level 2 cluster behavior uses a significantly different model than that of level 0 and level 1.

The level 2 behavior is easy to describe, but it may be difficult to understand in practical terms. In brief, level 2 performs no merging of clusters whatsoever.

This means that there is no initial base-and-mark merging step (as is done in level 0), and it means that reordering moves and ligature substitutions do not trigger a cluster merge.

Only one shaping operation directly affects clusters when using level 2:

- When a cluster *decomposes*, all of the resulting child clusters inherit as their cluster value the cluster value of the parent cluster.

When glyphs do form a ligature (or when some other feature substitutes multiple glyphs with one glyph) the cluster value of the first glyph is retained as the cluster value for the resulting ligature.

This occurrence sounds similar to a cluster merge, but it is different. In particular, no subsequent characters MDASH including marks and modifiers MDASH are affected. They retain their previous cluster values.

Level 2 cluster behavior is ultimately less complex than level 0 or level 1, but there are several cases for which processing cluster values produced at level 2 may be tricky.

Ligatures with combining marks in level 2

The first example of how HarfBuzz's level 2 cluster behavior can be tricky is when the text to be shaped includes combining marks attached to ligatures.

Let us start with an input sequence with the following characters (top row) and initial cluster values (bottom row):

```
A,acute,B,breve,C,circumflex
0,1    ,2,3    ,4,5
```

If the sequence `A,B,C` forms a ligature, then these are the cluster values HarfBuzz will return under the various cluster levels:

Level 0:

```
ABC,acute,breve,circumflex
0  ,0    ,0    ,0
```

Level 1:

```
ABC,acute,breve,circumflex
0  ,0    ,0    ,5
```

Level 2:

```
ABC,acute,breve,circumflex
0  ,1    ,3    ,5
```

Making sense of the level 2 result is the hardest for a client program, because there is nothing in the cluster values that indicates that `B` and `C` formed a ligature with `A`.

In contrast, the ”merged” cluster values of the mark glyphs that are seen in the level 0 and level 1 output are evidence that a ligature substitution took place.

Reordering in level 2

Another example of how HarfBuzz's level 2 cluster behavior can be tricky is when glyphs reorder. Consider an input sequence with the following characters (top row) and initial cluster values (bottom row):

```
A,B,C,D,E
0,1,2,3,4
```

Now imagine `D` moves before `B` in a reordering operation. The cluster values will then be:

```
A,D,B,C,E
0,3,1,2,4
```

Next, if `D` forms a ligature with `B`, the output is:

```
A,DB,C,E
0,3 ,2,4
```

However, in a different scenario, in which the shaping rules of the script instead caused `A` and `B` to form a ligature *before* the `D` reordered, the result would be:

```
AB,D,C,E
0 ,3,2,4
```

There is no way for a client program to differentiate between these two scenarios based on the cluster values alone. Consequently, client programs that use level 2 might need to undertake additional work in order to manage cursor positioning, text attributes, or other desired features.

Other considerations in level 2

There may be other problems encountered with ligatures under level 2, such as if the direction of the text is forced to the opposite of its natural direction (for example, Arabic text that is forced into left-to-right directionality). But, generally speaking, these other scenarios are minor corner cases that are too obscure for most client programs to need to worry about.

Utilities

HarfBuzz includes several auxiliary components in addition to the main APIs. These include a set of command-line tools, a set of lower-level APIs for common data types that may be of interest to client programs, and an embedded library for working with Unicode Character Database (UCD) data.

Command-line tools

HarfBuzz include three command-line tools: `hb-shape`, `hb-view`, and `hb-subset`. They can be used to examine HarfBuzz's functionality, debug font binaries, or explore the various shaping models and features from a terminal.

hb-shape

`hb-shape` allows you to run HarfBuzz's `hb_shape()` function on an input string and to examine the outcome, in human-readable form, as terminal output. `hb-shape` does *not* render the results of the shaping call into rendered text (you can use `hb-view`, below, for that). Instead, it prints out the final glyph indices and positions, taking all shaping operations into account, as if the input string were a HarfBuzz input buffer.

You can specify the font to be used for shaping and, with command-line options, you can add various aspects of the internal state to the output that is sent to the terminal. The general format is

```
hb-shape [OPTIONS]
  path/to/font/file.ttf
  yourinputtext
```

The default output format is plain text (although JSON output can be selected instead by specifying the option [--output-format=json]). The default output syntax reports each glyph name (or glyph index if there is no name) followed by its cluster value, its horizontal and vertical position displacement, and its horizontal and vertical advances.

Output options exist to skip any of these elements in the output, and to include additional data, such as Unicode code-point values, glyph extents, glyph flags, or interim shaping results.

Output can also be redirected to a file, or input read from a file. Additional options enable you to enable or disable specific font features, to set variation-font axis values, to alter the language, script, direction, and clustering settings used, to enable sanity checks, or to change which shaping engine is used.

For a complete explanation of the options available, run

```
hb-shape --help
```

hb-view

`hb-view` allows you to see the shaped output of an input string in rendered form. Like `hb-shape`, `hb-view` takes a font file and a text string as its arguments:

```
hb-view [OPTIONS]
path/to/font/file.ttf
yourinputtext
```

By default, `hb-view` renders the shaped text in ASCII block-character images as terminal output. By appending the `--output-file=filename` switch, you can write the output to a PNG, SVG, or PDF file (among other formats).

As with `hb-shape`, a lengthy set of options is available, with which you can enable or disable specific font features, set variation-font axis values, alter the language, script, direction, and clustering settings used, enable sanity checks, or change which shaping engine is used.

You can also set the foreground and background colors used for the output, independently control the width of all four margins, alter the line spacing, and annotate the output image with

In general, `hb-view` is a quick way to verify that the output of HarfBuzz's shaping operation looks correct for a given text-and-font combination, but you may want to use `hb-shape` to figure out exactly why something does not appear as expected.

hb-subset

`hb-subset` allows you to generate a subset of a given font, with a limited set of supported characters, features, and variation settings.

By default, you provide an input font and an input text string as the arguments to `hb-subset`, and it will generate a font that covers the input text exactly like the input font does, but includes no other characters or features.

```
hb-subset [OPTIONS]
path/to/font/file.ttf
yourinputtext
```

For example, to create a subset of Noto Serif that just includes the numerals and the lowercase Latin alphabet, you could run

```
hb-subset [OPTIONS]
NotoSerif-Regular.ttf
0123456789abcdefghijklmnopqrstuvwxyz
```

There are options available to remove hinting from the subsetted font and to specify a list of variation-axis settings.

Common data types and APIs

HarfBuzz includes several APIs for working with general-purpose data that you may find convenient to leverage in your own software. They include set operations and integer-to-integer mapping operations.

HarfBuzz uses set operations for internal bookkeeping, such as when it collects all of the glyph IDs covered by a particular font feature. You can also use the set API to build sets, add and remove elements, test whether or not sets contain particular elements, or compute the unions, intersections, or differences between sets.

All set elements are integers (specifically, `hb_codepoint_t` 32-bit unsigned ints), and there are functions for fetching the minimum and maximum element from a set. The set API also includes some functions that might not be part of a generic set facility, such as the ability to add a contiguous range of integer elements to a set in bulk, and the ability to fetch the next-smallest or next-largest element.

The HarfBuzz set API includes some conveniences as well. All sets are lifecycle-managed, just like other HarfBuzz objects. You increase the reference count on a set with `hb_set_reference()` and decrease it with `hb_set_destroy()`. You can also attach user data to a set, just like you can to blobs, buffers, faces, fonts, and other objects, and set destroy callbacks.

HarfBuzz also provides an API for keeping track of integer-to-integer mappings. As with the set API, each integer is stored as an unsigned 32-bit `hb_codepoint_t` element. Maps, like other objects, are reference counted with reference and destroy functions, and you can attach user data to them. The mapping operations include adding and deleting integer-to-integer key:value pairs to the map, testing for the presence of a key, fetching the population of the map, and so on.

There are several other internal HarfBuzz facilities that are exposed publicly and which you may want to take advantage of while processing text. HarfBuzz uses a common `hb_tag_t` for a variety of OpenType tag identifiers (for scripts, languages, font features, table names, variation-axis names, and more), and provides functions for converting strings to tags and vice-versa.

Finally, HarfBuzz also includes data type for Booleans, bit masks, and other simple types.

UCDN

HarfBuzz includes a copy of the UCDN (Unicode Database and Normalization) library, which provides functions for accessing basic Unicode character properties, performing canonical composition, and performing both canonical and compatibility decomposition.

Currently, UCDN supports direct queries for several more character properties than HarfBuzz's built-in set of Unicode functions does, such as the BiDirectional Class, East Asian Width, Paired Bracket and Resolved Linebreak properties. If you need to access more properties than HarfBuzz's internal implementation provides, using the built-in UCDN functions may be a useful solution.

The built-in UCDN functions are compiled by default when building HarfBuzz from source, but this can be disabled with a compile-time switch.

Platform Integration Guide

HarfBuzz was first developed for use with the GNOME and GTK software stack commonly found in desktop Linux distributions. Nevertheless, it can be used on other operating systems and platforms, from iOS and macOS to Windows. It can also be used with other application frameworks and components, such as Android, Qt, or application-specific widget libraries.

This chapter will look at how HarfBuzz fits into a typical text-rendering pipeline, and will discuss the APIs available to integrate HarfBuzz with contemporary Linux, Mac, and Windows software. It will also show how HarfBuzz integrates with popular external libraries like FreeType and International Components for Unicode (ICU) and describe the HarfBuzz language bindings for Python.

On a GNOME system, HarfBuzz is designed to tie in with several other common system libraries. The most common architecture uses Pango at the layer directly "above" HarfBuzz; Pango is responsible for text segmentation and for ensuring that each input `hb_buffer_t` passed to HarfBuzz for shaping contains Unicode code points that share the same segment properties (namely, direction, language, and script, but also higher-level properties like the active font, font style, and so on).

The layer directly "below" HarfBuzz is typically FreeType, which is used to rasterize glyph outlines at the necessary optical size, hinting settings, and pixel resolution. FreeType provides APIs for accessing font and face information, so HarfBuzz includes functions to create `hb_face_t` and `hb_font_t` objects directly from FreeType objects. HarfBuzz can use FreeType's built-in functions for font_funcs vtable in an `hb_font_t`.

FreeType's output is bitmaps of the rasterized glyphs; on a typical Linux system these will then be drawn by a graphics library like Cairo, but those details are beyond HarfBuzz's control. On the other hand, at the top end of the stack, Pango is part of the larger GNOME framework, and HarfBuzz does include APIs for working with key components of GNOME's higher-level libraries MDASH most notably GLib.

For other operating systems or application frameworks, the critical integration points are where HarfBuzz gets font and face information about the font used for shaping and where HarfBuzz gets Unicode data about the input-buffer code points.

The font and face information is necessary for text shaping because HarfBuzz needs to retrieve the glyph indices for particular code points, and to know the extents and advances of glyphs. Note that, in an OpenType variable font, both of those types of information can change with different variation-axis settings.

The Unicode information is necessary for shaping because the properties of a code point (such as its General Category (gc), Canonical Combining Class (ccc), and decomposition) can directly impact the shaping moves that HarfBuzz performs.

GNOME integration, GLib, and GObject

As mentioned in the preceding section, HarfBuzz offers integration APIs to help client programs using the GNOME and GTK framework commonly found in desktop Linux distributions.

GLib is the main utility library for GNOME applications. It provides basic data types and conversions, file abstractions, string manipulation, and macros, as well as facilities like memory allocation and the main event loop.

Where text shaping is concerned, GLib provides several utilities that HarfBuzz can take advantage of, including a set of Unicode-data functions and a data type for script information. Both are useful when working with HarfBuzz buffers. To make use of them, you will need to include the `hb-glib.h` header file.

GLib's Unicode manipulation API includes all the functionality necessary to retrieve Unicode data for the unicode_funcs structure of a HarfBuzz `hb_buffer_t`.

The function `hb_glib_get_unicode_funcs()` sets up a `hb_unicode_funcs_t` structure configured with the GLib Unicode functions and returns a pointer to it.

You can attach this Unicode-functions structure to your buffer, and it will be ready for use with GLib:

```
#include <hb-glib.h>
...
hb_unicode_funcs_t *glibufunctions;
glibufunctions = hb_glib_get_unicode_funcs();
hb_buffer_set_unicode_funcs(buf, glibufunctions);
```

For script information, GLib uses the `GUnicodeScript` type. Like HarfBuzz's own `hb_script_t`, this data type is an enumeration of Unicode scripts, but text segments passed in from GLib code will be tagged with a `GUnicodeScript`. Therefore, when setting the script property on a `hb_buffer_t`, you will need to convert between the `GUnicodeScript` of the input provided by GLib and HarfBuzz's `hb_script_t` type.

The `hb_glib_script_to_script()` function takes an `GUnicodeScript` script identifier as its sole argument and returns the corresponding `hb_script_t`. The `hb_glib_script_from_script()` does the reverse, taking an `hb_script_t` and returning the `GUnicodeScript` identifier for GLib.

Finally, GLib also provides a reference-counted object type called `GBytes` that is used for accessing raw memory segments with the benefits of GLib's lifecycle management. HarfBuzz provides a `hb_glib_blob_create()` function that lets you create an `hb_blob_t` directly from a `GBytes` object. This function takes only the `GBytes` object as its input; HarfBuzz registers the GLib `destroy` callback automatically.

The GNOME platform also features an object system called GObject. For HarfBuzz, the main advantage of GObject is a feature called GObject Introspection. This is a middleware facility that can be used to generate language bindings for C libraries. HarfBuzz uses it to build its Python bindings, which we will look at in a separate section.

FreeType integration

FreeType is the free-software font-rendering engine included in desktop Linux distributions, Android, ChromeOS, iOS, and multiple Unix operating systems, and used by cross-platform programs like Chrome, Java, and GhostScript. Used together, HarfBuzz can perform shaping on Unicode text segments, outputting the glyph IDs that FreeType should rasterize from the active font as well as the positions at which those glyphs should be drawn.

HarfBuzz provides integration points with FreeType at the face-object and font-object level and for the font-functions virtual-method structure of a font object. To use the FreeType-integration API, include the `hb-ft.h` header.

In a typical client program, you will create your `hb_face_t` face object and `hb_font_t` font object from a FreeType `FT_Face`. HarfBuzz provides a suite of functions for doing this.

In the most common case, you will want to use `hb_ft_font_create_referenced()`, which creates both an `hb_face_t` face object and `hb_font_t` font object (linked to that face object), and provides lifecycle management.

It is important to note, though, that while HarfBuzz makes a distinction between its face and font objects, FreeType's `FT_Face` does not. After you create your `FT_Face`, you must set its size parameter using `FT_Set_Char_Size()`, because an `hb_font_t` is defined as an instance of an `hb_face_t` with size specified.

```
#include <hb-ft.h>
...
FT_New_Face(ft_library, font_path, index, &face);
FT_Set_Char_Size(face, 0, 1000, 0, 0);
hb_font_t *font = hb_ft_font_create(face);
```

`hb_ft_font_create_referenced()` is the recommended function for creating an `hb_face_t` face object. This function calls `FT_Reference_Face()` before using the `FT_Face` and calls `FT_Done_Face()` when it is finished using the `FT_Face`. Consequently, your client program does not need to worry about destroying the `FT_Face` while HarfBuzz is still using it.

Although `hb_ft_font_create_referenced()` is the recommended function, there is another variant for client code where special circumstances make it necessary. The simpler version of the function is `hb_ft_font_create()`, which takes an `FT_Face` and an optional destroy callback as its arguments. Because `hb_ft_font_create()` does not offer lifecycle management, however, your client code will be responsible for tracking references to the `FT_Face` objects and destroying them when they are no longer needed. If you do not have a valid reason for doing this, use `hb_ft_font_create_referenced()`.

After you have created your font object from your `FT_Face`, you can set or retrieve the load_flags of the `FT_Face` through the `hb_font_t` object. HarfBuzz provides `hb_ft_font_set_load_flags()` and `hb_ft_font_get_load_flags()` for this purpose. The ability to set the load_flags through the font object could be useful for enabling or disabling hinting, for example, or to activate vertical layout.

HarfBuzz also provides a utility function called `hb_ft_font_has_changed()` that you should call whenever you have altered the properties of your underlying `FT_Face`, as well as a `hb_ft_get_face()` that you can call on an `hb_font_t` font object to fetch its underlying `FT_Face`.

With an hb_face_t and hb_font_t both linked to your FT_Face, you will typically also want to use FreeType for the font_funcs vtable of your hb_font_t. As a reminder, this font-functions structure is the set of methods that HarfBuzz will use to fetch important information from the font, such as the advances and extents of individual glyphs.

All you need to do is call

```
    hb_ft_font_set_funcs(font);
```

and HarfBuzz will use FreeType for the font-functions in font.

As we noted above, an hb_font_t is derived from an hb_face_t with size (and, perhaps, other parameters, such as variation-axis coordinates) specified. Consequently, you can reuse an hb_face_t with several hb_font_t objects, and HarfBuzz provides functions to simplify this.

The hb_ft_face_create_referenced() function creates just an hb_face_t from a FreeType FT_Face and, as with hb_ft_font_create_referenced() above, provides lifecycle management for the FT_Face.

Similarly, there is an hb_ft_face_create() function variant that does not provide the lifecycle-management feature. As with the font-object case, if you use this version of the function, it will be your client code's respsonsibility to track usage of the FT_Face objects.

A third variant of this function is hb_ft_face_create_cached(), which is the same as hb_ft_face_create() except that it also uses the generic field of the FT_Face structure to save a pointer to the newly created hb_face_t. Subsequently, function calls that pass the same FT_Face will get the same hb_face_t returned MDASH and the hb_face_t will be correctly reference counted. Still, as with hb_ft_face_create(), your client code must track references to the FT_Face itself, and destroy it when it is unneeded.

Uniscribe integration

If your client program is running on Windows, HarfBuzz offers an additional API that can help integrate with Microsoft's Uniscribe engine and the Windows GDI.

Overall, the Uniscribe API covers a broader set of typographic layout functions than HarfBuzz implements, but HarfBuzz's shaping API can serve as a drop-in replacement for Uniscribe's shaping functionality. In fact, one of HarfBuzz's design goals is to accurately reproduce the same output for shaping a given text segment that Uniscribe produces MDASH even to the point of duplicating known shaping bugs or deviations from the specification MDASH so you can be confident that your users' documents with their existing fonts will not be affected adversely by switching to HarfBuzz.

At a basic level, HarfBuzz's `hb_shape()` function replaces both the `ScriptShape()` and `ScriptPlace()` functions from Uniscribe.

However, whereas `ScriptShape()` returns the glyphs and clusters for a shaped sequence and `ScriptPlace()` returns the advances and offsets for those glyphs, `hb_shape()` handles both. After `hb_shape()` shapes a buffer, the output glyph IDs and cluster IDs are returned as an array of hb__glyph__info__t structures, and the glyph advances and offsets are returned as an array of hb__glyph__position__t structures.

Your client program only needs to ensure that it coverts correctly between HarfBuzz's low-level data types (such as `hb_position_t`) and Windows's corresponding types (such as `GOFFSET` and `ABC`). Be sure you read the Buffers, language, script and direction chapter for a full explanation of how HarfBuzz input buffers are used, and see Shaping and buffer output for the details of what `hb_shape()` returns in the output buffer when shaping is complete.

Although `hb_shape()` itself is functionally equivalent to Uniscribe's shaping routines, there are two additional HarfBuzz functions you may want to use to integrate the libraries in your code. Both are used to link HarfBuzz font objects to the equivalent Windows structures.

The `hb_uniscribe_font_get_logfontw()` function takes a `hb_font_t` font object and returns a pointer to the `LOGFONTW` "logical font" that corresponds to it. A `LOGFONTW` structure holds font-wide attributes, including metrics, size, and style information.

The `hb_uniscribe_font_get_hfont()` function also takes a `hb_font_t` font object, but it returns an `HFONT` MDASH a handle to the underlying logical font MDASH instead.

`LOGFONTW`s and `HFONT`s are both needed by other Uniscribe functions.

As a final note, you may notice a reference to an optional `uniscribe` shaper back-end in the Configuration options section of the HarfBuzz manual. This option is not a Uniscribe-integration facility.

Instead, it is a internal code path used in the `hb-shape` command-line utility, which hands shaping functionality over to Uniscribe entirely, when run on a Windows system. That allows testing HarfBuzz's native output against the Uniscribe engine, for tracking compatibility and debugging.

Because this back-end is only used when testing HarfBuzz functionality, it is disabled by default when building the HarfBuzz binaries.

Core Text integration

If your client program is running on macOS or iOS, HarfBuzz offers an additional API that can help integrate with Apple's Core Text engine and the underlying

Core Graphics framework. HarfBuzz does not attempt to offer the same drop-in-replacement functionality for Core Text that it strives for with Uniscribe on Windows, but you can still use HarfBuzz to perform text shaping in native macOS and iOS applications.

Note, though, that if your interest is just in using fonts that contain Apple Advanced Typography (AAT) features, then you do not need to add Core Text integration. HarfBuzz natively supports AAT features and will shape AAT fonts (on any platform) automatically, without requiring additional work on your part. This includes support for AAT-specific TrueType tables such as `mort`, `morx`, and `kerx`, which AAT fonts use instead of `GSUB` and `GPOS`.

On a macOS or iOS system, the primary integration points offered by HarfBuzz are for face objects and font objects.

The Apple APIs offer a pair of data structures that map well to HarfBuzz's face and font objects. The Core Graphics API, which is slightly lower-level than Core Text, provides `CGFontRef`, which enables access to typeface properties, but does not include size information. Core Text's `CTFontRef` is analagous to a HarfBuzz font object, with all of the properties required to render text at a specific size and configuration. Consequently, a HarfBuzz `hb_font_t` font object can be hooked up to a Core Text `CTFontRef`, and a HarfBuzz `hb_face_t` face object can be hooked up to a `CGFontRef`.

You can create a `hb_face_t` from a `CGFontRef` by using the `hb_coretext_face_create()`. Subsequently, you can retrieve the `CGFontRef` from a `hb_face_t` with `hb_coretext_face_get_cg_font()`.

Likewise, you create a `hb_font_t` from a `CTFontRef` by calling `hb_coretext_font_create()`, and you can fetch the associated `CTFontRef` from a `hb_font_t` font object with `hb_coretext_face_get_ct_font()`.

HarfBuzz also offers a `hb_font_set_ptem()` that you an use to set the nominal point size on any `hb_font_t` font object. Core Text uses this value to implement optical scaling.

When integrating your client code with Core Text, it is important to recognize that Core Text `points` are not typographic points (standardized at 72 per inch) as the term is used elsewhere in OpenType. Instead, Core Text points are CSS points, which are standardized at 96 per inch.

HarfBuzz's font functions take this distinction into account, but it can be an easy detail to miss in cross-platform code.

As a final note, you may notice a reference to an optional `coretext` shaper back-end in the Configuration options section of the HarfBuzz manual. This option is not a Core Text-integration facility.

Instead, it is a internal code path used in the `hb-shape` command-line utility, which hands shaping functionality over to Core Text entirely, when run on a macOS system. That allows testing HarfBuzz's native output against the Core Text engine, for tracking compatibility and debugging.

Because this back-end is only used when testing HarfBuzz functionality, it is disabled by default when building the HarfBuzz binaries.

ICU integration

Although HarfBuzz includes its own Unicode-data functions, it also provides integration APIs for using the International Components for Unicode (ICU) library as a source of Unicode data on any supported platform.

The principal integration point with ICU is the `hb_unicode_funcs_t` Unicode-functions structure attached to a buffer. This structure holds the virtual methods used for retrieving Unicode character properties, such as General Category, Script, Combining Class, decomposition mappings, and mirroring information.

To use ICU in your client program, you need to call `hb_icu_get_unicode_funcs()`, which creates a Unicode-functions structure populated with the ICU function for each included method. Subsequently, you can attach the Unicode-functions structure to your buffer:

```
hb_unicode_funcs_t *icufunctions;
icufunctions = hb_icu_get_unicode_funcs();
hb_buffer_set_unicode_funcs(buf, icufunctions);
```

and ICU will be used for Unicode-data access.

HarfBuzz also supplies a pair of functions (`hb_icu_script_from_script()` and `hb_icu_script_to_script()`) for converting between ICU's and HarfBuzz's internal enumerations of Unicode scripts. The `hb_icu_script_from_script()` function converts from a HarfBuzz `hb_script_t` to an ICU `UScriptCode`. The `hb_icu_script_to_script()` function does the reverse: converting from a `UScriptCode` identifier to a `hb_script_t`.

By default, HarfBuzz's ICU support is built as a separate shared library (`libharfbuzz-icu.so`) when compiling HarfBuzz from source. This allows client programs that do not need ICU to link against HarfBuzz without unnecessarily adding ICU as a dependency. You can also build HarfBuzz with ICU support built directly into the main HarfBuzz shared library (`libharfbuzz.so`), by specifying the `--with-icu=builtin` compile-time option.

Python bindings

As noted in the GNOME integration, GLib, and GObject section, HarfBuzz uses a feature called GObject Introspection (GI) to provide bindings for Python.

At compile time, the GI scanner analyzes the HarfBuzz C source and builds metadata objects connecting the language bindings to the C library. Your

Python code can then use the HarfBuzz binary through its Python interface.

HarfBuzz's Python bindings support Python 2 and Python 3. To use them, you will need to have the `pygobject` package installed. Then you should import `HarfBuzz` from `gi.repository`:

```
from gi.repository import HarfBuzz
```

and you can call HarfBuzz functions from Python. Sample code can be found in the `sample.py` script in the HarfBuzz `src` directory.

Do note, however, that the Python API is subject to change without advance notice. GI allows the bindings to be automatically updated, which is one of its advantages, but you may need to update your Python code.

HB_VERSION=2.6.8