The Apple Font Tool Suite Tutorial

Table of Contents

Introduction

This tutorial is a general introduction to the Mac OS X Apple Font Tool Suite, illustrating the various techniques needed to work with a real-life font.

Further documentation on each of the tools can be found in the Apple Font Tool Suite document and the Quick Reference, both of which are installed by the installer. There is also a text file, `'Tutorial Command Summary.txt'`, which contains all the command lines found in this Tutorial and is handy for cutting and pasting into the terminal window.

The tutorial includes a basic font, `Apple Simple.ttf`, which was created by Apple for demonstration purposes only.

The tutorial can be usefully worked through in conjunction with a general font editor such as ***Fontographer***, ***RoboFog***, or ***FontLab***. However, if you do not have a glyph editing tool, then begin each lesson with the ready-made versions of the Apple Simple font which you will find in each lesson folder. The results of each command execution are also included so that, if necessary, the tutorial can be read and studied without running the exercises live. All files have unique two digit prefixes to enable identification of any file across all Tutorial folders. E.g. "`05_GlyphPalette.pdf`". This convention facilitates reading, note-making and greatly helps rapid roll-back when needed.

The tutorial assumes general familiarity with Unix, XML and Unicode: you need to know how to launch the Unix ***Terminal*** application, how command-line utilities work, and how to navigate from one folder to another using the ***cd*** command. Note that if you pre-edit and store command line instructions in a word processor, you need to de-activate any smart quote or autoformat substitution so that ASCII quote marks are not changed to curly ones and the dash is not changed to em-dash as these are not recognized by the command line shell. The em-dash can even cause a paste operation to scramble and the cursor to jump. If this happens, go back and retype any dashes in a plain text editor.

For XML, the reader should be familiar with HTML tagging and understand the basics of how XML is structured. We use the standard `U+XXXX` notation for Unicode code points and `<U+XXXX U+YYYY>` for sequences of Unicode code points. Adobe's alternative standard notation `uniXXXX` is also used in places.

We strongly recommend *The Unicode Standard, Version 3.0* as an invaluable reference for identifying and naming glyphs, and learning about the scope of Unicode multi-script support and bi-directional line layout. Less formal books on Unicode include *Unicode: A Primer* by Tony Graham and *Unicode Demystified* by Richard Gillam. If you don't have the book, then for glyph repertoire planning work, you can still download PDF copies of individual code blocks from the book from the Unicode web site at `http://www.unicode.org/charts/`. The Unicode standard data tables are also available

on the site at http://www.unicode.org/Public/UNIDATA/ as well as other useful information at http://www.unicode.org/Public/.

It's important to be clear on the distinction between characters and glyphs: characters are units of text storage and processing. Operations such as searching and sorting use characters. Glyphs are visual shapes representing marks, signs or symbols. Glyphs are usually given names in the 'post' table (aka "postnames"). A character mapping (or "cmap") will provide the default information on how to go between them. Fonts can have multiple cmaps in their 'cmap' table, allowing the font to be used with more than one character set (such as MacRoman and Unicode).

The relationship between characters and glyphs is usually one-to-one, but it can be more complex. That is, a glyph can have zero, one or more mapping entries in a cmap; and conversely a cmap entry may map to the undefined glyph (".notdef", which is as close as an entry can be to zero short of not existing), or to a single glyph (the norm). In addition to its one mapped glyph, a character may have many associated display variant glyphs. The 'cmap' table does not directly support one-to-many character-to-glyph mappings. Instead, the extra associated display variant glyphs are selected not by cmap but by use of style and typographic feature tags in the application's UI or automatically by surrounding glyph context.

Consequently, in fonts there will be some glyphs that do not have Unicode cmap entries, and a few that may have multiple entries. Only you as the font designer can determine which these should be. Tools cannot help at this stage. Tools can only help once you have established accurate primary glyph identifiers in the form of either postnames or cmap entries.

To get to this state, you need to choose your primary glyph identifier, carefully check the values and then derive the secondary identifier from the primary one. The primary-secondary distinction is only to determine where you invest your verification checking work and the sequence of generation. Once both 'cmap' and 'post' tables are created and checked, the distinction has no significance.

The choice of whether the cmap entries or the glyph names are your primary identifiers depends on the kind of font you have. In the case of East Asian fonts, which have no glyph names, it has to be the cmap. In a font with 1-to-1 character-to-glyph mappings, the primary identifier can also be the cmap. In a font with many display variants which are unencoded (i.e., do not have cmap entries), it is generally better to work through the postnames first as your primary identifiers, as these are the superset, and then generate the cmap entries from the glyph names. For fonts with display variants articulated by shaping behaviors, the postnames are also needed in writing the shaping behavior source files (MIF) and the optional kerning (KIF) and justification source files (JIF).

Careful glyph naming is therefore needed to support this process, and we recommend use of glyph names from the Apple Glyph Name database (part of this tool suite) for encoded glyphs. For unencoded glyphs we recommend the Adobe glyph naming

conventions: use of the period "." and underscore "_"delimiters to structure the name strings of unencoded glyphs so that they are of the form `<encodedglyph name>+"."+<variantname>` , e.g. `"ampersand.oldstyle4"; "f_i.terminalswash.3"`. Use of postnames from the Apple Glyph Name database ensures that ***ftxanalyzer*** and ***ftxenhancer*** will recognize the glyphs and process them correctly. Use of the delimited name suffix format enables you to group associated glyphs together in name hierarchies that may be parsed by future tools. The Adobe naming convention spec is available at http://partners.adobe.com/asn/developer/type/unicodegn.html.

Apart from this introduction, the tutorial does not cover principles of font design, nor how to create a font with commercial font editors. We assume normal font production and debugging has already been completed, i.e., that the contents of the font are correct. In particular, the ***ftxanalyzer*** tool assumes that it is analyzing a font with correct postnames and Unicode cmap entries.

The focus of the tutorial is on enhancement rather than debugging, i.e., on how Apple's suite of font tools can be used to add new features for Mac OS X and Unicode support. Along the way though, the methods given also show how you can debug fonts by fixing important basic postname and cmap problems in an already existing font created by a commercial font editor.

Note that all these tools automatically *overwrite* pre-existing files of the same name without warning. Therefore, if you want to keep previous file versions, take care to specify different output filenames or copy previous versions out of the folder before execution. As always, never work on your only copy of any file.

On commercial editors: fonts created with ***Fontographer*** will require more attention than will fonts created with ***FontLab***, as ***Fontographer 4.1.x*** has not been significantly revised since 1995 and allows little control over generation of postnames, encodings, and other features. ***RoboFog*** is a ***Fontographer 3.5***-derived tool that gives good control via Python scripting. ***FontLab*** is the most comprehensive glyph and font editing tool we know of, and—as of this writing (October 2002)—is in release 4.5 for Windows, Mac OS 9, and Mac OS X.

Lesson One: Filling Out the Glyph Repertoire

OK, the font is designed: You've either created your basic glyphs with ***Fontographer***, ***RoboFog*** or ***FontLab***, or you have a raw font file from elsewhere and you're ready to go. What next?

The first step is to review and fill any "holes" that may exist in the glyph repertoire. How could "holes" exist in your font when you, the designer, have carefully chosen and drawn all the glyphs you want?

The answer is through the expectations placed upon a font, firstly by computer hardware, and secondly by the newly emerging software requirements of a Unicode-based world, in which a far wider variety of data sequences must be handled by the font even in so-called "ordinary" situations. These new expectations stem from three sources:

(1) Macintosh script repertoires for cmap synchronization.

MacRoman has been the standard 8-bit encoding for Macintosh since 1985. The MacRoman character sets therefore defines the minimal repertoire that a Mac font should contain as all these characters can be typed from a normal Mac keyboard. Absence of any of the Mac Roman glyphs in your font would cause a square ".notdef" glyph to appear when the user typed that character, or, if the Mac OS X font-fallback is supported by the application, the substitution of a glyph from another font which usually results in style mismatches and a 'ransom note' effect.

Another important aspect of the MacRoman encoding legacy is that a mixed environment of both MacRoman and Unicode-based applications is a reality that we will be living with for some time. Therefore cmap synchronization is important so that data created in Classic or Carbon applications which use the MacRoman character set can be rendered by a Unicode-based application and then saved back as MacRoman without loss or corruption. This means that every MacRoman cmap entry should have a corresponding entry in the Unicode cmap, and also that these two entries should be mapped to the same glyph. Lack of synchronization of the two cmaps could lead to the same text appearing differently in MacRoman and Unicode-based applications, e.g., MacRoman characters missing from the Unicode cmap would appear as `.notdef` boxes in Unicode-based applications.

The same is, of course, true for any other Macintosh character set your font is intended to support. If, for example, yours is a Japanese font covering the MacJapanese character set, this should be consistent with its Unicode coverage.

Mappings between Apple's legacy character sets and Unicode can be found at http://www.unicode.org/Public/MAPPINGS/VENDORS/APPLE/.

(2) Decomposed Unicode support.

Many accented Latin letters can be represented in Unicode in two different ways: either as a single *precomposed* character, or as a sequence of the base letter followed by one or more *composing* accent characters.

For example, the 'e-acute' letter é can be represented either as the precomposed e-acute character U+00E9 or as the ordinary **e** character U+0065 followed by the combining acute accent character ´ U+0301 .

Unicode recommends that any application support both forms and treat them the same. Different applications may use different forms. For example, the Mac OS X file system decomposes all filenames to enable consistent string searching and sorting. Other applications may use precomposed forms of Unicode rather than decomposed forms, and still others may use a mixture. Because of this, your font has to handle both forms of text representation. The impact on a font's repertoire is usually that the combining forms of the accents need to be added.

(3) Bi-directional rendering support.

The third source of additional demands on a font repertoire come from the fact that with Unicode, a font can be used in text containing scripts that run in opposite directions; so-called bi-directional text rendering. The two most common right-to-left scripts are Arabic and Hebrew. In these situations some characters reverse their direction—so-called horizontal mirroring. E.g. "(" turns to ")", "[" to "]", and so on. The consequence for your font's glyph repertoire is that you should have both members of any mirroring glyph pair.

(4) Vertical text rendering.

A fourth source of glyph requirements is vertical text rendering, where some glyphs rotate by 90° or change their position in order to work with vertically presented Chinese, Japanese, and Korean (CJK) texts. However, as this is normally covered by the standard repertoires of CJK fonts and there are few occurrences of vertical text rendering outside of CJK-specific applications, this feature is not covered by the present suite of font tools.

Automatic repertoire analysis with *ftxanalyzer*

The task of reconciling your font repertoire against the requirements of the older Macintosh character sets plus all the possible Unicode decompositions plus all mirroring glyphs is a complex and very time-consuming task if done manually. We have therefore automated the process on Mac OS X by creation of ***ftxanalyzer***. This tool examines a font's cmap and post table contents, does all the above checking and makes recommendations based on what it finds. The recommendations are made in the form of text files that you can examine, edit and use as input to other tools, ***ftxenhancer***

and ***ftxdumperfuser***, which will actually compile changes into the font. So let's start by analyzing the contents of a sample font, Apple Simple:

Launch the ***Terminal*** application and navigate to the folder for the first tutorial lesson. The Mac OS X Terminal has a handy shortcut to help you do this: If you drag-and-drop a file or folder icon onto the Terminal window it will enter its path name into the command line. In this case, first type `cd<space>`, then drop the `"Lesson 1"` folder onto the window and hit return. You will have now changed the current directory ("`cd`") and be inside the lesson one folder. You can confirm this through listing the contents of your current location by typing `ls –l` or just `ls`.

Analyzing a font's Unicode cmap for creation of an add list

Now execute the following command line:

```
ftxanalyzer -g 02_AppleSimple.add 01_Apple\ Simple.ttf
```

Let's look at this line point-by-point:

"`ftxanalyzer`" is the command we're executing. Everything else is an option telling it what to do. This is the same as launching a Mac application and choosing options from the menus.

The "`-g`" is an option telling ***ftxanalyzer*** to generate a file of glyphs (hence the "`g`") that it thinks should be added to the font based on examination of the Unicode cmap. This file is called an "add list" and it can be used by ***ftxenhancer*** to actually add new glyphs to the font. The `-g` option takes an argument, meaning that what follows after the `-g` is the name of the add file to be created in the current folder—in this case "`02_AppleSimple.add`".

Finally, we have the font file name. Notice it contains a back-slash ("\"). Ordinarily, Unix uses spaces to separate words on a command line. For this reason, spaces in file names are usually avoided when working with Unix. However, since the name of the font file has a space in it, we need to do something to keep the Unix shell from thinking this is two arguments, "`01_Apple`" and "`Simple.ttf`". The backslash is a special "escape character" which tells Unix that the following space is part of the argument and not an argument separator. We could also achieve the same thing by using single ASCII quotes around the string instead of an escape character in front of every space. E.g.

```
ftxanalyzer -g 02_AppleSimple.add '01_Apple Simple.ttf'
```

Examining the add list

If we open the add file we've just created in a text editor such as TextEdit or BBEdit, we see there are twenty-six entries. The format of an add file is described fully in the Apple Font Tool Suite document. The 26 entries divide into a group of fourteen and a group of twelve. These are for decomposed Unicode support and mirroring respectively.

Add list—decomposed Unicode analysis The initial fourteen entries are nonspacing accents. These are needed for decomposed Unicode support. ***ftxanalyzer*** attempts to ensure that any precomposed Unicode character can also be represented in the composing form. That is, if it finds a precomposed character such as `U+00E9 (eacute)` in your font, it checks to make sure that the decomposed components are also present. In the case of eacute, the decomposed forms would be `U+0065 (e)` and `U+0301 (acutecombining)`. ***ftxanalyzer*** uses the suffixes "`cmb`" for "`combining`" on a glyph name, as you can see in the first two add list entries shown below:

```
gravecmb     1      //     U+0300 COMBINING GRAVE ACCENT
grave 0      0
acutecmb     1      //     U+0301 COMBINING ACUTE ACCENT
acute 0      0
```

In our case, ***ftxanalyzer*** has found fourteen combining accents that are needed for the composing equivalent of precomposed characters in the font. It therefore recommends that they be added. For these fourteen, ***ftxanalyzer*** has found another glyph in the font that could possibly be copied to handle the situation. E.g. for the combining grave accent, it suggests the non-combining grave accent, and so on. Only well-known glyphs have built-in suggestions. In the absence of a suggestion the default is to use the undefined glyph "`.notdef`" as a place-holder in the add list. It is then the designer's job to replace the empty `.notdef` box with a suitable glyph. Ideally, the combining accents never occur in isolation, so it shouldn't really matter what they look like, but `.notdef` is chosen as a default so that if they do, and if you don't otherwise provide a glyph for them, it looks to the end user like an unsupported character instead of nonsense. You as the font designer must choose one of three ways to handle combining glyphs:

Combining Option (1)—Zero-width reverse offset: Give glyphs a zero width and a negative left side-bearing. This makes them hang over the previous character so they can combine with suitably-sized and positioned glyphs. Given the variation in glyph shape, width and position, this will always be a compromise for many combinations and you will have to choose a trade-off. Designers generally optimize for lower case composing (e.g. with "e"), as lower case is the most frequent case. Alternatively, if you already have pre-composed glyphs for all the lower case combinations you are expecting in your target language(s), then it may be more useful to position the glyph to combine with upper case letters in order to broaden the coverage of your design.

Combining Option (2)—Spacing: Give glyphs a positive width and normal side-bearings. This makes them easily visible when they show up in isolation. As theoretically this should not happen, it can be considered a bug in the data when it does happen. Therefore, some people advocate highlighting this by putting a dotted circle with the glyph in the style of the Unicode book so you *know* that you've got a combining accent. An example of this is given for Pollard tone marks at the end of Lesson Four.

Combining Option (3)—Invisible: Make the glyph a copy of the ".null" glyph which has an empty path and zero-width. This means that if the character occurs in isolation,

it will be invisible in the text and will not alter the text metrics. The downside to this is it will also be impossible to detect by normal visual inspection and the text data will therefore go uncorrected. Invisible characters usually interfere with the string matching processes of searching and sorting, so think carefully before choosing this design option.

Add list—bi-directional rendering analysis The remaining twelve glyphs ***ftxanalyzer*** recommends be added have to do with mirroring in Unicode. Most languages, when written horizontally, run left-to-right across the line, but some, such as Hebrew and Arabic, are written right-to-left. Support for these scripts and for the complex situations where bi-directional scripts are intermingled is an important part of Unicode.

Some characters mirror in right-to-left contexts, that is, the glyphs for them flip horizontally. E.g. "(" turns to ")", "[" to "]", and so on. Most of these characters already come in pairs in Unicode, but not all do. The exceptions are mostly mathematical characters. It is these that ***ftxanalyzer*** has identified in the Apple Simple font. The first four entries of the add list are shown below:

```
notequal_mirror    1
notequal    1065   0      -X -1.0      -Y 1.0
notequal_clone     1
notequal    0      0
integral_mirror    1
integral    656    0      -X -1.0      -Y 1.0
integral_clone     1
integral    0      0
```

The six Unicode characters in the font which should have mirror image glyphs added are: "`notequal`" (≠), "`approxequal`" (≈), "`integral`" (∫), "`radical`" (√), "`summation`" (∑), and "`partialdiff`" (∂). These are all part of the standard MacRoman encoding .

For each mirrored glyph required, ***ftxanalyzer*** has added two glyphs: the mirrored glyph and a copy of the original glyph labeled with a "`_clone`" suffix. This is a workaround for the fact that most font editors don't enable control of glyph order, yet the base and mirrored glyph need to be close to each other in the font.

We could add all twenty-six glyphs in the default add list that ***ftxanalyzer*** generated (i.e. like in option (3) above), but this is usually not a good idea. There are two reasons.

One has to do with the flow of work in making a font. If you're ever going to go back and add new glyphs in your ***FontLab***, ***Fontographer***, or ***RoboFog*** sources, then it will get confusing for you if you have to add yet more glyphs (again) using ***ftxenhancer***. Best to use only one process for adding new glyphs if you can.

The other reason is that ftxanalyzer will do its best to make the new glyph out of pieces of old glyphs. It can even mirror glyphs as needed—but it will fall back on .notdef if it

can't do anything else. The `.notdef` glyph is glyph 0 and looks like a hollow box ( ). This is definitely *not* how mirrored mathematical symbols should appear, and is questionable for anything else. Beyond that, there may be subtle features of your font design that don't work well with ***ftxanalyzer***'s defaults. The mirror glyph for the left-to-right integral sign may not look exactly like the mirror image of the glyph. You, the font designer, can make that decision best.

The various ways of adding glyphs to the font are described later in this Tutorial.

Add list—cmap round-trip analysis Before we add the glyphs, let's use one more analyzer option relating to repertoire analysis to check that there are no other glyphs that may need to be created. Type:

```
ftxanalyzer -M -g 03_AppleSimple.add '01_Apple Simple.ttf'
```

Now compare the output file to the previous one i.e. `03_AppleSimple.add` and `02_AppleSimple.add`. Either just open it up and have a look or use the Unix utility, "`diff`" which is very useful for comparing large files. To do the latter, type the following line:

```
diff 02_AppleSimple.add 03_AppleSimple2.add
```

Don't concern yourself with interpreting the screen output. Just note that there is a long list of deltas so the two files are *very* different: There are, in fact, 26 additional entries at the top of the add list. The first two are shown below:

```
controlSOT   1      //    U+0002 <control>
.notdef      0      0
controlETX   1      //    U+0003 <control>
.notdef      0      0
```

The `-M` option tells ***ftxanalyzer*** to analyze any Mac cmaps in the font to make sure that every character in them is also in the Unicode cmap. ***ftxanalyzer*** expects each cmap entry to be mapped to a glyph, i.e., that all the glyphs needed to support full round-trip compatibility are present. However, it does not check that the glyphs match.

Apple Simple has a standard MacRoman cmap. The 26 extra add list entries that ***ftxanalyzer***'s `–M` option found are the MacRoman control characters `0x00` through `0x1F` and `0x7F`, which map to `U+0000` through `U+001F` and `U+007F` in Unicode. They have no glyph mappings. ***ftxanalyzer*** therefore adds 26 entries to the add list as a suggestion that we do something about it.

In this case, we can ignore ***ftxanalyzer***'s advice as there is no pressing reason to have Unicode's C0 range control characters covered by the font. The only ones that are likely to occur in a text file are tab, carriage-return, and line-feed, and applications generally don't draw these characters. As the unused control characters always turn up in the analysis, the mapping analysis option is split from the `–g` option to avoid cluttering the add list. However, occasionally the mapping analysis will show up a

wanted marking character that has dropped out of the Unicode cmap. It is therefore worth running the –M option separately to ensure that the Mac and Unicode cmaps are synchronized (except for control characters).

Apple Simple happens to only have MacRoman, but fonts can contain multiple cmaps (called "code pages" in Windows) for support of other languages and scripts (e.g. MacHebrew, MacCyrillic, MacThai, etc.). If we didn't want a massive add list back from all the possible unmapped entries in these extra encodings, we could limit ***ftxanalyzer*** to only consider one of them by using the **-l** option as well as the **-M** option. This limits the mapping analysis to the Mac encoding cmap indicated by the 'FOND' resource ID value.

Adding the new mirrored glyphs to the font

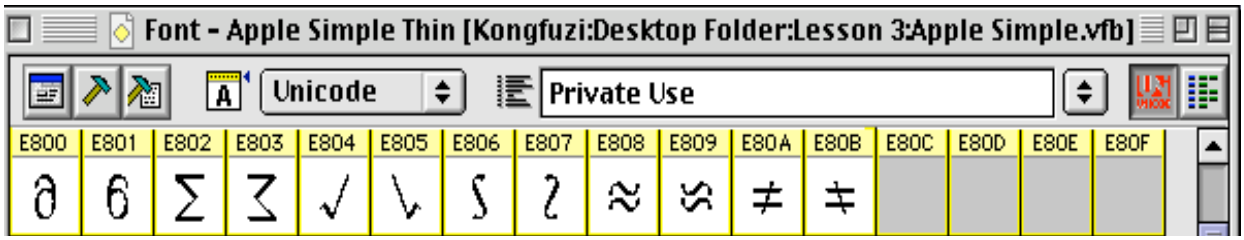
OK, so let's add the glyphs! What's the best way to do it and what are the requirements?

The goal is to end up with the six new mirrored glyphs in the font *next to* their original unmirrored forms. The "next to" is important as it is presently a Mac OS X system requirement for Unicode bi-directional text support that mirrored glyphs are *within 7* glyph index entries from their base forms in the font. This requirement comes from the present design of the 'prop' table format. Only seven slots leeway is as good as being tied together so we recommend you adopt the convention of always placing the glyphs next to each other to help in checking.

Normally glyph sequence doesn't matter beyond the first 4 standard glyphs (.notdef, .null, nonmarkingreturn and space) in a Macintosh TrueType font. Mirroring is an additional special case for Unicode support that we're highlighting in this tutorial.

In the case of Apple Simple, the six new mirrored glyphs in the add list have to end up next to their original base forms. This is shown in the name listing and glyph palette screen-shot from FontLab below:

```
notequal
notequal_mirror
approxequal
approxequal_mirror
integral
integral_mirror
radical
radical_mirror
summation
summation_mirror
partialdiff
partialdiff_mirror
```



The general technique for each glyph is to create a new empty glyph and then copy the base glyph outline and flip it horizontally. This is done by the Add list or it can also be done in a font editing application.

The tricky part is controlling the TrueType glyph index sequence, i.e. the physical sequence of the outline glyphs in the 'glyf' table, which is the source of the glyph index values and arrangement of the so-called "glyph palette".

Some font tools don't allow you to do this directly, so different methods are needed depending on which font editor you use:

Fontographer 4.x doesn't give you control of glyph palette arrangement.

RoboFog probably does via Python scripting.

FontLab 3.x doesn't give direct control of glyph palette arrangement but has a TrueType preferences option called "Use Unicode indexes as a base for TrueType encoding" which sorts the glyph indexes in ascending Unicode order. You can use this option with suitably faked Unicode values in the Private Use Area (`U+E000` onwards) followed by clean-up of the Unicode cmap with ***ftxdumperfuser*** after generating the TrueType file.

FontLab 4.x gives direct viewing and control of the glyph palette arrangement. This is used in combination with the TrueType preferences option called "Do not reorder glyphs" so as to preserve glyph index sequence when generating the font.

Manually adding the glyphs in FontLab

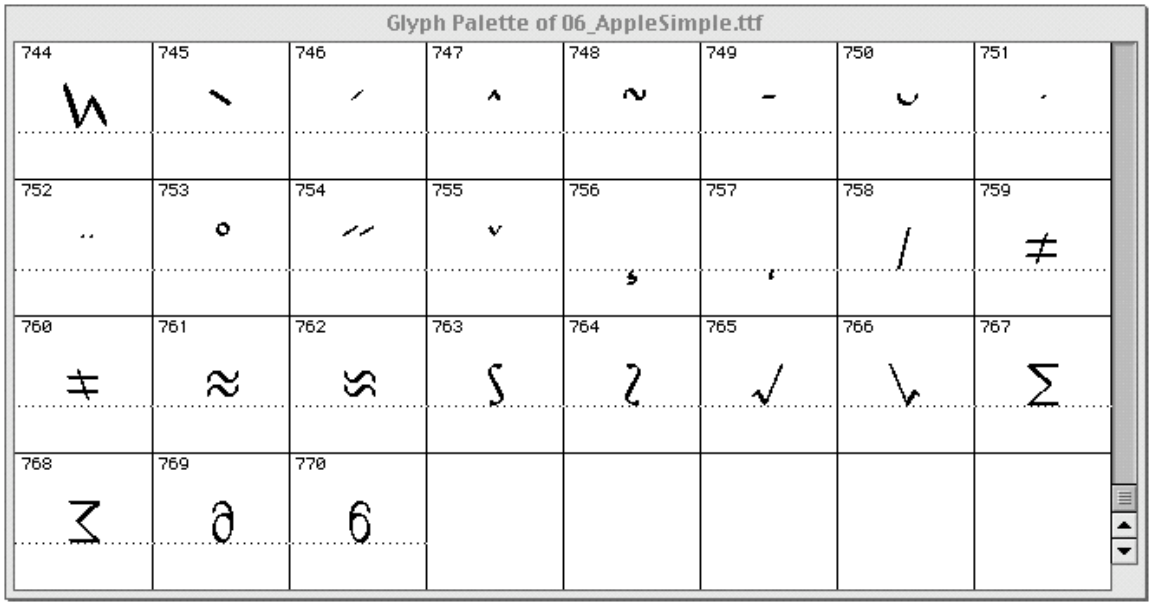
Assuming you are working in FontLab 4.5, you simply create the six additional glyphs, copy, paste and flip the outlines for them, then click on "view by glyph index" mode and drag glyphs around to re-arrange the font's glyph sequence so that the mirrored pairs are next to each other. Below is the font after flipping the mirrored glyphs in FontLab.

By the way, the glyph palette screen-shot below is from ***TrueEdit***—one of the free downloadable tools from `http://developer.apple.com/fonts/` that can be run in the Mac OS X 10.2 Classic environment.

TrueEdit is handy for viewing the physical glyph palette in index order and verifying the results of glyph editing operations and font re-generation (use the "Glyph Palette…" option which is **Command-L** in the **Edit** menu). TrueEdit requires that the

Mac file type code to be set to `'sfnt'` before it can open a .ttf font. You can set this with ***ResEdit***'s **Get File/Folder info**... option or any other Mac file type code editing utility such as ***FileTyper***. It can also be set in ***Terminal*** using the command `/Developer/Tools/SetFile -t sfnt` followed by the file name (you must have installed the Developer CD for this command to be available).

TrueEdit also provides a good printed version of the glyph palette, an example of which is included in the Lesson One folder in the file `05_GlyphPalette.pdf`, which shows the full glyph palette of the font at this stage of editing.



The font file that represents this stage of editing is `06_AppleSimple.ttf`

Do working checks after each major file modification

Having modified the font in ***FontLab*** and then saved and re-generated the TrueType file, there is the possibility that errors occurred in edit actions or that the .ttf was generated with inappropriate options (e.g. glyphs accidentally rearranged in Unicode order). Therefore, it is always good practice to check the font before moving on with more table work. Things you can do:

(1) ***TrueEdit*** glyph palette inspection, which will tell you if the glyphs got re-arranged on re-generation.

(2) Run ***ftxvalidator***. The test is fast and prevents easily caught errors creeping in. If you get a lot of extraneous warning and information messages, you can reduce this to

just the error messages by using the –r e option. Look for any differences in the errors reported between the before and after versions of the font.

```
ftxvalidator 01_AppleSimple.ttf
ftxvalidator 06_AppleSimple.ttf

ftxvalidator –r e 06_AppleSimple.ttf
```

(3) Run ***ftxanalyzer*** repertoire analysis if you modified the Unicode cmap.

```
ftxanalyzer -g 07_AppleSimple.add 06_AppleSimple.ttf
```

This will ensure firstly that no obvious errors were made in the cmap assignments and secondly that no additional repertoire requirements for Unicode were introduced. Note that as this is a cmap test, it will only detect whether the cmap entry has a glyph assigned or not and will not check if the actual name of the glyph is appropriate. So check the glyph names you add yourself carefully.

If you added all the correct glyphs with the correct Unicode code points, the add file should be empty. If you did not add Unicode values to the glyphs, then the add file should contain the same entries as before.

For edits to glyphs in the MacRoman encoding, also run the –M option to check that the cross-mapping is intact. Again, if all is well then the list should not have changed from before.

```
ftxanalyzer -M -g 08_AppleSimple.add 06_AppleSimple.ttf
```

Glyph index re-arrangement workaround for non-FontLab users

This is a detour into add lists which are not discussed fully until the next Lesson. You may want to return to this section after working through Lesson Two.

In the case of being *unable* to control the glyph index sequence directly in your font editing tool, a workaround is to use ***ftxenhancer*** and the add list generated from ***ftxanalyzer*** to create clones of the base glyphs next to each of the mirrored glyphs.

The add list section from `02_AppleSimple.add` that we are interested in looks like the following, with a clone glyph being created next to each mirror glyph. Using add lists is covered in more detail in Lesson Two.

```
notequal_mirror     1
notequal            1065 0     -X -1.0      -Y 1.0
notequal_clone      1
notequal            0    0
integral_mirror     1
integral            656  0     -X -1.0      -Y 1.0
integral_clone      1
integral            0    0
partialdiff_mirror  1
partialdiff         936  0     -X -1.0      -Y 1.0
partialdiff_clone   1
```

```
partialdiff          0    0
summation_mirror     1
summation            1180 0     -X -1.0      -Y 1.0
summation_clone      1
summation            0    0
radical_mirror       1
radical              1068 0     -X -1.0      -Y 1.0
radical_clone        1
radical              0    0
approxequal_mirror   1
approxequal          1272 0     -X -1.0      -Y 1.0
approxequal_clone    1
approxequal          0    0
```

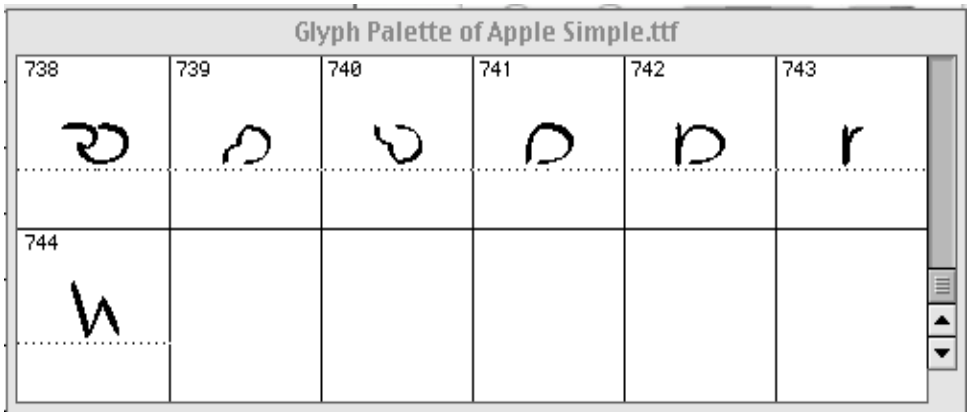
The mirror glyphs are flipped automatically by ftxenhancer because the X-scale parameter is negative (i.e. `-X -1.0`). The scaler supports -1 for the scale of a composite glyph, and ***ftxenhancer*** understands this too. The pair of numbers before the `-X` and `-Y` scale factors are the X and Y values to shift the glyph after the scaling—in this case, we shift it to the right by the glyph's width lest it end up flipped over its own left edge. In the default add list, ftxanalyzer has copied the width of the base glyph on the assumption that widths of mirrored glyphs should match, but you could edit this if necessary.

Note that a simple flip is OK for geometric typeform designs, but calligraphic letterforms usually need to be re-drawn to make the reversed shape consistent with actual pen swings and weight stress.

In the add list, the glyphs are given temporary suffix strings "`_mirror`" and "`_clone`" to distinguish them. After creation of the glyphs, the postnames and cmap entries are then edited to point to the cloned glyphs and their postname suffixes removed (e.g. "summation_clone" → "summation") so they match the original base glyph names.

At the same time, the original base glyphs, which are now deadwood in the font, are either deleted or given postnames that indicate this (e.g. "notequal" →"notequal.unused" or "unused3" etc.). The technique for cmap and postname cleanup is described fully in Lesson Three.

Below is a view of the end of Apple Simple's glyph palette before the new glyphs are added. You can see the font contains *744* glyphs.



Making Mac file backup copies

Copy the original font before running the manually prepared add list on the duplicate font file. Type:

```
cp '01_Apple Simple.ttf' 09_AppleSimple.ttf
```

The standard Unix copy (`cp`) command above only copies the data fork of a file and loses the Mac type and creator codes and the resource fork. If you are using ***TrueEdit*** to verify `.ttf` font file modifications, this is a nuisance. You can avoid it in two ways: Either make finder copies or, if you have run the Developer disk installer, you can use the "`CpMac`" command from `/Developer/Tools` which makes full Mac file copies. Type:

```
/Developer/Tools/CpMac '01_Apple Simple.ttf' 09_AppleSimple.ttf
```

The most visible difference between these two methods is that the file copied with `CpMac` preserves the font file icon whereas the `cp` copy only has a generic icon.

If you are familiar with Unix, you can add `/Developer/Tools` to your `$path` environment variable (`echo $path`) so you don't have to always type the path string and CpMac will work on its own. If you do this, then the line below will work:

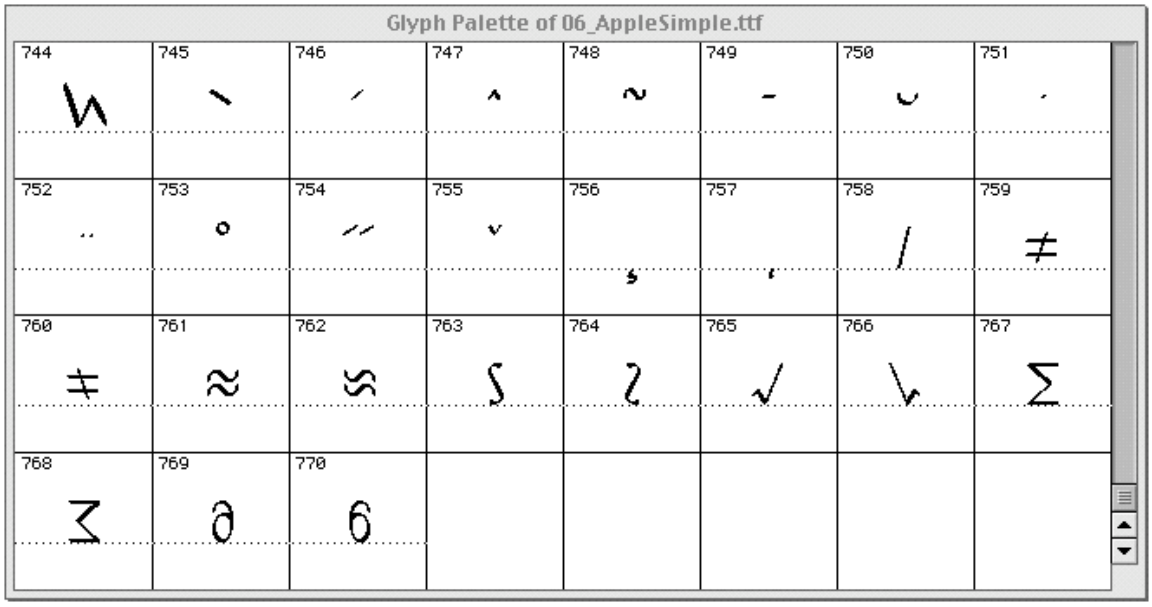
```
CpMac '01_Apple Simple.ttf' 09_AppleSimple.ttf
```

Run the add list to create the mirrored and cloned glyphs

```
ftxenhancer -A 02_AppleSimple.add 09_AppleSimple.ttf
```

Below is the glyph palette of `09_AppleSimple.ttf` after running the add list. You can see the glyph index extends from *744* to *770* with the 26 new glyphs added to the font.

Note that in order to minimize font size, *ftxenhancer* creates the cloned glyphs as composite references of the original glyph. You will therefore need to decompose each composite glyph before being able to edit (e.g. flip) it.



Add list as a batch glyph creation short-cut

In the case of having a font editing application that allows you to control glyph indexes, note that if you have many glyphs to add, that running an add list on the font is a handy way of creating the empty or cloned glyphs ready-named for you to edit the outline shapes. This can save time and reduce naming errors. See Lesson Two for more details.

Lesson Two: Using Add Lists

Having added glyphs the manual way with a font editing tool, we'll now use add lists to expand the glyph repertoire quickly and easily. Add lists are *very* useful when you want to add many characters to your font at once.

We'll start by adding the Roman numerals. These are defined in the Unicode block starting at U+2150. Navigate to the Lesson 2 folder (cd <drop folder on terminal>) and open the 10_Roman.add file. Let's examine it before we use it.

```
uni2160 1
I 0 0

uni2161 2 -C
I 0 0
I 0 0 -L

uni2162 3 -C
I 0 0
I 0 0 -L
I 1 0 -L

uni2163 2 -C
I 0 0
V 0 0 -L
```

As before, we have a number of entries (thirty-two, to be precise). Each consists of a number of lines. The first line starts with the glyph name (e.g. "uni2160") and number of lines to follow plus some flags (e.g. "-c"). This tells ***ftxenhancer*** what the name of the glyph is that we're going to make. Each of the subsequent lines in each group specifies one of the glyphs to use as a component for that particular new glyph and how to assemble the pieces.

The flags are explained in the add file section of the Apple Font Tool Suite document. We're not using many flags here: just the -c flag to tell *ftxenhancer* that the new glyphs have the accumulated width of the pieces, and the -L flag that tells it to assemble each component glyph next to the preceding one.

Note that the glyph names in this add list use Adobe's "uniXXXX" convention. This convention is handy if you don't have a set of descriptive glyph names to use. Instead you can simply look up the glyphs in *The Unicode Standard* and prefix each encoding value with "uni". If you don't have a copy of the Unicode book, you can still look up the Unicode values used in the Roman add list by downloading a PDF of the code chart of this block at www.unicode.org/charts/PDF/U2150.pdf. All Unicode code blocks are available from this site.

The uniXXXX convention also has the advantage of being understood by most font tools in current production. There's another good reason to do this, which is that ***ftxanalyzer*** can generate a cmap from the uniXXXX names with its –c option. This is shown later in this Lesson.

Handling tool errors

Let's now run the add list on our font. `11_AppleSimple.ttf` is a copy of `06_AppleSimple.ttf` from Lesson One:

```
ftxenhancer -A 10_Roman.add 11_AppleSimple.ttf
```

The tool failed! You get back the following complaints:

```
##### ERROR: Some other FontToolbox error occurred.
  -C
  Source file ftxframework/UtilityLibs/EnableMultiScriptFont/EMFErrors.cpp,
line 111
##### ERROR:  Aborting after failure to process AppleSimpleMirrL2.ttf.
```

What happened? The error message returned isn't terribly helpful. In such cases, run the tool again adding the **`-v`** (= "verbose") option to get more information on the error condition.

```
ftxenhancer -v -A 10_Roman.add 11_AppleSimple.ttf
```

This time we get a lot of running commentary on what is happening and we see right at the end that the tool was trying to add the glyph named "`uni216A`" from three pieces when the error happened. In general, the tools report when they are about to start an action rather than when it is completed. The error has occurred during creation of `uni216A` and before starting creation of the following entry, so you should look in the 216A entry to find the problem.

```
Making a new glyph named "uni2168" from 2 pieces
Making a new glyph named "uni2169" from 1 pieces
Making a new glyph named "uni216A" from 3 pieces
##### ERROR: Some other FontToolbox error occurred.
  -C
  Source file ftxframework/UtilityLibs/EnableMultiScriptFont/EMFErrors.cpp,
line 111
##### ERROR:  Aborting after failure to process AppleSimpleMirrL2.ttf.
```

Debugging the Add list

Time to re-examine the `10_Roman.add` file. Open it in a text editor and search for "`uni216A`". Before reading the answer in the next paragraph, look at other entries and refer to the add list format description in the Tool Suite document to learn the format and try to identify the error yourself.

It shouldn't take long to find the error: The first line for the `uni216A` group says that three pieces are following, when in fact only two follow. Use the Unicode code block chart to identify if a 3rd element is needed to fix the entry or whether the piece count should be reduced. You should now also double-check all the other piece counts as there are three more errors to be found.

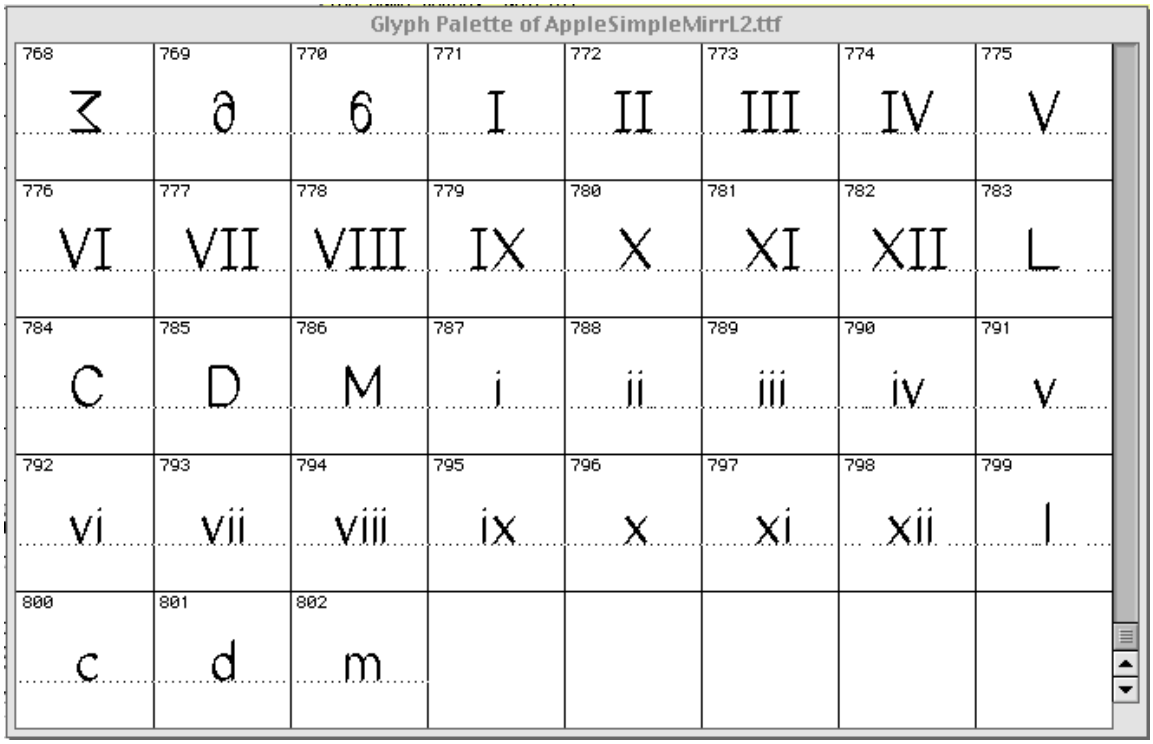
Note that if the tool aborts with an error, the font file is left untouched, so you can run the command repeatedly on the same font file until you get a clean execution. With all

Unix command line tools, "silence is success", meaning if the tool returns to the command line prompt without any message, it has executed successfully.

A handy Unix shell shortcut for re-running a command is to use the up- and down-arrow keys to scroll through all the previous commands. Once the chosen command is sitting on the line just hit return. For quick reference a corrected add list is in the file `12_Roman.add`.

```
ftxenhancer -A 12_Roman.add 11_AppleSimple.ttf
```

After successfully running the add list, if you now open the font in ***TrueEdit***, what you will see are the following 32 glyphs added to the end of the font, from ID 771 to 802:



Remember to close the font after viewing in ***TrueEdit***, otherwise the next tool you run may return an error since the font file is busy and can't be opened.

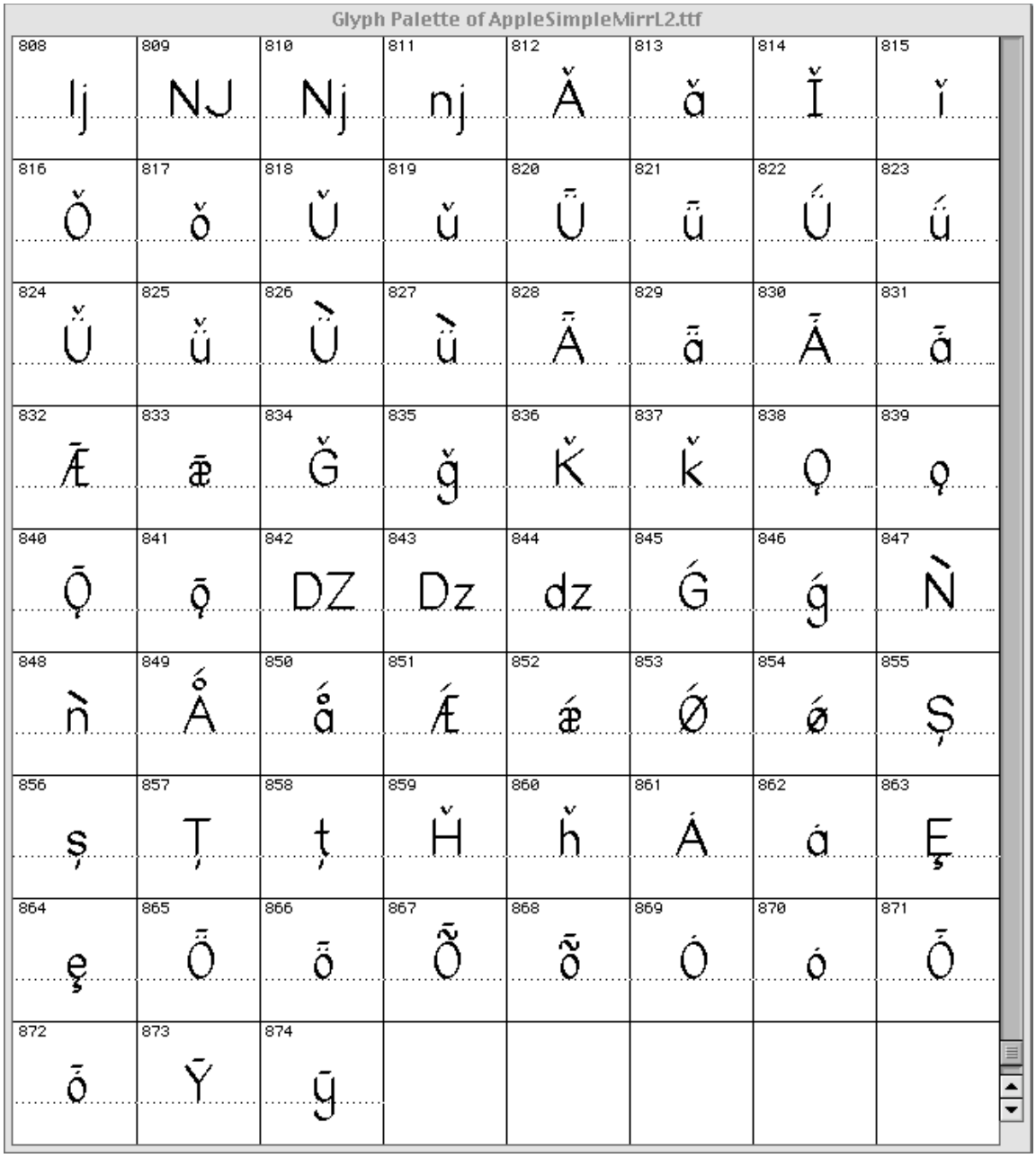
Running additional Add lists

You can build up a library of add lists which can be used on all your fonts, acting as skeleton frameworks for font repertoires. They can be re-used because Add lists use glyph names to specify the compositions rather than glyph indexes. Therefore as long as you use the glyph names in the database all your add lists can be applied to all your fonts. This has the benefit that once you have debugged an add list, you don't have to do the structural checking work again on each font.

As an example of how useful and fast these ready-made add lists can be, now run the `13_LatinExtensionB.add` file on the Apple Simple font. Verbose mode is not required

as this list executes OK. It adds 97 precomposed diacritic letters and ligatures, as shown in the glyph palette screen-shot below.

```
ftxenhancer -A 13_LatinExtensionB.add 11_AppleSimple.ttf
```



Add list checking

There are two levels to Add list correctness: Structure and Composition. Structure refers to the identity of the new glyph and the number, sequence, and identities of its component pieces. This comes down to checking that the glyph names are right. You don't have to check the piece count manually because the tool will report run-time errors on this, as we have already seen.

Composition refers to the relative positioning and overall metrics of the glyph. This has two parts: Firstly checking that the flags in the Add list are correctly set so that the components are assembled automatically as well as can be specified. Secondly checking that the glyph looks good and deciding which glyphs need additional manual adjustment of the positions of the shapes and the glyph metrics. Many diacritic compositions can usually benefit from adjusting the size of the diacritics.

```
uni01C6      3
d      0     0
z      0     0     -L
caron  3     5     -Z    -R1   -G

uni01E0      3
A      0     0
dotaccent    3     5     -Z    -G
macron       3     5     -Z    -R1   -G

uni01F0      2
dotlessj     0     0
caron  3     5     -Z    -G
```

All these aspects of checking can be done most efficiently by working through a ***TrueEdit*** print-out of the glyph palette and comparing each glyph to the Unicode chart glyph. We can do the Unicode comparison at this stage even though there is no Unicode cmap for these glyphs because the glyph names are of the `uniXXXX` form that contains the Unicode value, and the ***TrueEdit*** print-out displays the glyph name in each cell. Correcting Unicode codepoint errors at this stage halves the work as there are no cmap entries to correct.

Working on a paper print-out is also advised because you can mark any errors and then use the document as a check-list for the subsequent rounds of corrections.

The Latin Extension B add list used above is known to contain some design rather than run-time errors. To find out what to correct in `13_LatinExtensionB.add`, download the LatinExtensionB code block chart from the Unicode web site at http://www.unicode.org/charts/PDF/U0180.pdf and compare a print-out of the Unicode block chart with a print-out of `14_GlyphPalette.pdf` which contains the TrueEdit glyph palette of `11_AppleSimple.ttf`.

The corrected add list is in the file `15_LatinExtensionB.add`

Roll back the font and re-run the Add lists

As there are errors in glyph composition, we need to roll back to the previous saved version of the font which is before the LatinExtensionB glyphs were added. In this case it is `06_AppleSimple.ttf` from Lesson One (A copy is in the Lesson Two folder). If in doubt as to the contents of the repertoire, inspect it in ***TrueEdit***. Make a copy of the font and re-run the two add lists on the new copy. If in your work you didn't keep previous font versions to roll back to, you would have to open the font and correct any glyphs and postnames manually in a font editor.

Incrementing the filename may seem a chore, but it saves a great deal of confusion in the long run and lays a trail of intermediate file versions that can be invaluable for roll-backs. Preparing the command strings with the revised file names in advance helps do this—as below, where you can see the add list commands have been revised to use the corrected add list `15_LatinExtensionB.add` and operate on `16_AppleSimple.ttf`.

```
/Developer/Tools/CpMac 06_AppleSimple.ttf 16_AppleSimple.ttf

ftxenhancer -A 12_Roman.add 16_AppleSimple.ttf

ftxenhancer -A 15_LatinExtensionB.add 16_AppleSimple.ttf
```

Creating cmap entries for the new glyphs

OK, now we need to update the Unicode cmap so it has correct code point entries for each of the new glyphs.

You can do this manually in ***FontLab***, selecting each glyph in turn and typing in the code point. However, we've just added 156 glyphs, so typing in each by hand will take a long time and, as it is a very repetitive action, there is a high chance of human error.

If you're a `perl` guru, you could also write a quick `perl` script to do it using the add files as a source. There is, however, an even easier way: ***ftxanalyzer*** has a **-c** option that creates a cmap for a font. Execute:

```
ftxanalyzer -c 17_AppleSimple.cmap.xml 16_AppleSimple.ttf
```

To see what has just been done, open `AppleSimple.cmap.xml` with your text editor and look for "`uni2160`"—the name we gave to the Roman numeral "I".

```
<map charValue="0x2160" charName="ROMAN NUMERAL ONE" glyphRefID="771"
      glyphName="uni2160" />
```

ftxanalyzer found it, understood the "`uniXXXX`" glyph name convention to be a Unicode value, saw it didn't have a mapping in the existing Unicode cmap, and so added an entry to the XML output file. Ditto for the other 155 glyphs. This is because ***ftxanalyzer*** recognizes the "`uni`" prefix of the glyph name and will add a Unicode entry directly determined by the `XXXX` part of the name.

Glyph Name Database

ftxanalyzer also uses a database of known standard glyph names and their Unicode mappings. If you're curious, the glyph name database can be found inside `FTX.framework` in the file `Contents/Resources/FTXDatabase.xml`, the first few entries of which are shown below (the location and contents of this file are subject to change).

```
<characterDatabase date="2002.05.02" version="1.0">
<charData name=".notdef" vChar="0x0F0000" properties="11" flags="0xFFFE" />
<charData name=".null" vChar="0x000000" properties="0" flags="0xFFFE" />
<charData name="A" vChar="0x000041" properties="0" flags="0xFFFE" />
<charData name="AE" vChar="0x0000C6" properties="0" flags="0xFFFE" />
<charData name="AEacute" vChar="0x0001FC" properties="0" flags="0xFFFE" />
```

If you want to use descriptive names in an add list rather than the '`uniXXXX`' names, you can verify that you are using the right glyph names by opening a copy of the database file in a text editor and searching for your proposed glyph name. If your spelling does not exist, try a reverse lookup by searching on the Unicode four-digit hex to locate the entry and lookup the expected glyph name spelling.

The name database is updated periodically with new glyph names in the course of Apple's font production work. As fonts have a very long lifetime, old glyph names are not removed, but are kept as synonyms to a given Unicode code point. Therefore, some Unicode values have multiple names associated with them. You can email additional database glyph names to fonttools@apple.com. If the mappings and spellings do not conflict with existing entries, they may be added to a future revision of the database.

Note: do not attempt to edit the glyph name database. A future version of the font tool suite may provide the ability to specify an auxiliary glyph name database.

Prepare to Merge old and new cmap data

If we wanted to, we *could* just go ahead and add this '`cmap`' to the font using ***ftxdumperfuser***. It's a little risky to do this, however, because ***ftxdumperfuser*** can only completely replace the entire '`cmap`' table in the font and if there were any other cmap subtables in the font, such as MacRoman, they would be lost.

The way to avoid this problem is to dump the existing cmap table using ***ftxdumperfuser,*** then cut and paste the newly generated Unicode cmap file into it so that any other cmap subtables are preserved in the file and will be fused back into the font along with the new one.

An even more conservative approach is to paste the new individual character entries into the dump of the old Unicode cmap. This will preserve any hand-edited entries that might be in the cmap, such as multiple cmap entries pointing to the same glyph, which is something that the tools don't automatically re-generate (e.g., the same glyph "Omega" being mapped to both the Omega and Ohm characters).

To dump the old cmap type:

```
ftxdumperfuser -t cmap -o 18_AppleSimpleOrig.cmap.xml
      16_AppleSimple.ttf
```

The **-t** option tells ***ftxdumperfuser*** to work with the 'cmap' table, and the **-o** option tells it to output the dumped data into the file `18_AppleSimpleOrig.cmap.xml`. We now have an XML file with the old 'cmap' data as well as the previously generated new cmap in `17_AppleSimple.cmap.xml`. Open both files with a text editor.

cmap XML structure—where to cut and paste

XML is an HTML-like way of organizing data. The file consists of nodes, each of which has optional subnodes and optional attribute data. If a node has no subnodes, it terminates with a "/>" sequence. Otherwise, it has the name of the node within pointed brackets ("<" and ">") to start it, and then the same node name within "</" and ">" to end it.

A cmap XML file consists, at the top level, of a single `cmapTable` node. It has a `versionMajor` attribute and a `versionMinor` attribute, and one or more `cmapSubtable` nodes. Each `cmapSubtable` has platform, script, and language attributes, which are specified both with numbers and human-readable names. The cmap XML structure looks like this:

```
<cmapTable versionMajor="1" versionMinor="0">

      <cmapSubtable encodingID="0"
            format="0"
            platformID = "1"
            platformName="Macintosh"
            scriptID="0"
            scriptName="Roman"
            languageID="-1"
            languageName="No language"
            >
      […the Macintosh/Roman cmap data…]
      </cmapSubtable>

      <cmapSubtable encodingID="1"
            format="4"
            platformID = "3"
            platformName="Microsoft"
            scriptID="1"
            scriptName="Unicode"
            languageID="-1"
            languageName="No language"
            >
      […the Microsoft/Unicode cmap data…]
      […the Microsoft/Unicode cmap data…]
      […the Microsoft/Unicode cmap data…]   ← paste entries in here
      </cmapSubtable>
```

```
</cmapTable>
```

You can see that this 'cmap' table has two subtables—one for MacRoman and one for Microsoft/Unicode. A cmap could also have a Unicode/Unicode subtable.

We don't know if there are hand-edited cmap entries in this cmap, so to be safe we'll copy the lines for the new glyph mappings from the newly generated cmap file and paste them into the the old cmap file in the Microsoft/Unicode 'cmap' subtable. If there were a Unicode /Unicode 'cmap' subtable, we would also copy them into there.

Identify the new glyph entries in the ftxanalyzer cmap

Notice a difference in arrangement between the two cmap files: The newly generated cmap file created by *ftxanalyzer* is made by first copying the existing cmap and then going through each glyph in turn and looking up its postname. Any new glyphs are added in glyph sequence at the end of the file, which is a convenient place to find them. By contrast, the cmap created by ftxdumperfuser is arranged in ascending order by Unicode code point.

Copy the set of deltas from the new cmap `17_AppleSimple.cmap.xml` into a new text file. This is in order to do a check before adding them to the old cmap file. Use a line number or line count function in your word processor to find out how many new cmap entries you have. Then open `14_GlyphPalette.pdf`, which displays the glyph palette of `11_AppleSimple.ttf`, and count the new glyphs. The two counts should match. If they don't, go back and find the other cmap entries in `17_AppleSimple.cmap.xml`. The deltas you should have found are stored for reference in the text file `19_NewGlyphs.cmap.xml`.

If a cmap entry for a new glyph can't be found, it is an indication that there may be a postname error in the font and that ***ftxanalyzer*** has failed to recognize that glyph name and so not created a cmap entry. If this is the case, dump the post table and inspect the glyph names to find the offending string; correct it; fuse the post table back and re-run the cmap analysis or manually add the entries.

```
ftxdumperfuser -t post -p -o 20_AppleSimple.post.xml 16_AppleSimple.ttf
```

On inspection of the post table dump, you will notice that the mirrored mathematics glyphs are not included. This is because the "_mirror" and "_clone" suffixes are not recognized by the present version of *ftxanalyzer*. They will be dealt with manually in Lesson 3.

```
<PostScriptName glyphRefID="759" NameString="notequal_clone" />
<PostScriptName glyphRefID="760" NameString="notequal_mirror" />
<PostScriptName glyphRefID="761" NameString="approxequal_clone" />
<PostScriptName glyphRefID="762" NameString="approxequal_mirror" />
<PostScriptName glyphRefID="763" NameString="integral_clone" />
<PostScriptName glyphRefID="764" NameString="integral_mirror" />
<PostScriptName glyphRefID="765" NameString="radical_clone" />
<PostScriptName glyphRefID="766" NameString="radical_mirror" />
<PostScriptName glyphRefID="767" NameString="summation_clone" />
<PostScriptName glyphRefID="768" NameString="summation_mirror" />
<PostScriptName glyphRefID="769" NameString="partialdiff_clone" />
```

```
<PostScriptName glyphRefID="770" NameString="partialdiff_mirror" />
```

Pasting in the new cmap entries

Select the glyphs from `19_NewGlyphs.cmap.xml` and paste them into the bottom of the Unicode cmap subtable in the location described above. The entries can be placed out of Unicode sequence at the end of the file as *ftxdumperfuser* will sort the file before compiling it into the font. Don't worry about the fact that the deltas have an additional Unicode character name field, either, as this attribute is ignored by ***ftxdumperfuser***. The new cmap file is saved as `21_AppleSimple.cmap.xml`.

Some manual checking still needed

The Font Tool Suite does not currently provide a function for reconciliation of the glyphs' postnames with their cmap entries (i.e. to check that glyphs of the right name are assigned to each Unicode point), nor for detecting hand-edited deltas in the Unicode cmap (i.e. multiple cmap entries that map to the same glyph). Therefore, you will have to ensure accurate glyph names and cmap assignments and keep track of any glyphs that are mapped more than once.

Making a font backup copy

Before we fuse the result back in, it's a very good idea to make a copy of the font. If anything goes wrong, you can roll back easily, and having the copy lets you do a comparison afterwards. You can copy the font file either in the Finder or in Terminal by using the `cp` or `CpMac` command.

We suggest you "copy forward" rather than simply making a file labeled "_old" or "_bak", i.e., you take the latest version of the font and copy it forward to a higher version in which you continue working. In this way, the backup copies are labeled by their natural sequence in the work history, which is more useful for identification later on. In this example, file #16 is copied forward to #22, which is the 22nd working file in this Tutorial. Of course, you can use any naming convention that you like.

```
/Developer/Tools/CpMac 16_AppleSimple.ttf 22_AppleSimple.ttf
```

Fusing the cmap into the font

Now we can use ***ftxdumperfuser*** to fuse the new data back into the font:

```
ftxdumperfuser -t cmap -d 21_AppleSimple.cmap.xml 22_AppleSimple.ttf
```

Again, the `-t` option tells *ftxdumperfuser* to work on the '`cmap`' table, and the `-d` option tells it to take data from the file `21_AppleSimple.cmap.xml` and fuse it into the font.

Working check on cmap fuse

Run a working check on this operation immediately by dumping the table back out to verify that the new entries were fused in successfully. Include postnames (-p) in the dump to help identify the new glyphs.

```
ftxdumperfuser -t cmap -p -o 23_AppleSimple.cmap.xml 22_AppleSimple.ttf
```

As before, search for "uni2160," the Roman numeral "I," and verify that (a) it exists and (b) that it is pointing at the right glyph index (GID=771), which you can look up on the glyph palette print-out.

Font change comparison with *ftxdiff*

Now that we've made a successful change to the 'cmap' data, we should also confirm that no other, unexpected changes have been made. We do this using ***ftxdiff***. We've already mentioned `diff`, the general Unix utility to compare two versions of a file. ***ftxdiff*** does the equivalent thing with two versions of a TrueType™ or OpenType™ font. It will analyze two fonts for semantic differences and list them.

```
ftxdiff -o 24_AppleSimple2216.dif 22_AppleSimple.ttf 16_AppleSimple.ttf
```

The filename `24_AppleSimple2216.dif` is specified as the argument to the output (**–o**) option and will now contain the diff report. Open it in a text editor and inspect it. Since we didn't do anything other than add some new mappings, *ftxdiff* should list the new mappings but otherwise not indicate any change except to the `'head'` table checksum for the entire font.

Do working checks after each major file modification

As always, we should review which working checks are needed after our modifications—in this case we've added two new batches of glyphs and their associated Unicode values using ***ftxenhancer*** and ***ftxdumperfuser***.

(1) *TrueEdit*—The glyph palette does not need inspection, as there is no danger of rearrangement with ***ftxenhancer***, which always adds glyphs at the end.

(2) Run ***ftxvalidator***. Look for any differences in the errors reported between the before and after versions of the font. Here we are comparing errors present before the new glyphs and cmap entries were added to now. The desirable result is that both error reports are the same, indicating that no new defects have been introduced by our work.

```
ftxvalidator 06_AppleSimple.ttf
ftxvalidator 22_AppleSimple.ttf
```

(3) Run ***ftxanalyzer***. We modified the Unicode cmap so a repertoire analysis is good to do again to make sure that there aren't any new gaps in the repertoire needed for full Unicode support.

```
ftxanalyzer -g 25_AppleSimple.add 22_AppleSimple.ttf
```

The add file should contain the same number of entries as the previous one, so compare `07_AppleSimple.add` with `25_AppleSimple.add`. This is best done using the Unix `diff` command, i.e.

```
diff 07_AppleSimple.add 25_AppleSimple.add
```

Sure enough, the `diff` report throws up two lines that are different: `25_AppleSimple.add` has one more entry than before. We will need to add this glyph, call it `commaaccent`, and map it to `U+0326`.

```
commaaccent 1      //    U+0326 COMBINING COMMA BELOW
.notdef     0      0
```

Note that in this case ***ftxanalyzer*** hasn't suggested a base glyph, such as `comma`, but instead has used the default glyph called "`.notdef`" (for "not defined"). You will have to create the combining comma below glyph yourself by copying and/or editing an outline using your font editing tool.

(4) Run ***ftxdiff***. This has just been introduced above, and is a very good way to certify that only the changes you intended have happened.

Exercises left for the reader:

(1) Add the combining comma below glyph for `U+0326` and save the result in `26_AppleSimple.ttf`. Do this either manually or by running an add list. You could use a copy of the comma rather than `.notdef`. If using an add list, look-up the right add list flags to position the glyph in the middle of an advance width and below the baseline. If you're not sure what the glyph should look like, download the PDF from the Unicode web site at http://www.unicode.org/charts/PDF/U0300.pdf .

(2) Try the other add list in the Tutorial Files folder, `27_LatinExtAdditional.add`, which contains 245 entries. Make a forward copy of the font to experiment on:

```
/Developer/Tools/CpMac 22_AppleSimple.ttf 28_AppleSimple.ttf
```

Then run the add list and see what happens:

```
ftxenhancer -A 27_LatinExtAdditional.add  28_AppleSimple.ttf
```

If you need to undo changes to the font, simply re-run the copy command line above by using the up-arrow short-cut in Terminal and the file `28_AppleSimple.ttf` will be overwritten with a fresh copy of #22.

Inspect the glyph palette in TrueEdit. A printout of `28_AppleSimple.ttf` is in the file `29_AppleSimple.pdf`. Are there any structural errors in this add list? How many glyphs need composition refinements?

After adding the glyphs, what is the impact on the repertoire requirements for full Unicode support?

Lesson Three: Completing the tables

Now it's time for us to complete the definitions of the six new mirrored mathematical glyphs we added. `cd` to the Lesson Three folder.

As outlined in Lesson One, we want to change the clones of the six math glyphs into the live glyphs and deprecate the original ones. To do this, we'll have to make changes in two places: first the font's `'post'` table (where glyph names live), and then the font's `'cmap'` table (where characters are mapped to glyphs).

Editing the Post table clone entries

Dump the post table:

```
ftxdumperfuser -t post -o 30_AppleSimple.post.xml 22_AppleSimple.ttf
```

The `'post'` table lists the name of each glyph in the font and is arranged in glyph index order. You can therefore verify the physical arrangement of the glyphs in the font by inspection of the `'post'` table instead of by viewing the glyph palette (assuming the glyph names are correct).

Recall from earlier that the goal for the mirrored math characters is to have all the "`_clone`" and "`_mirror`" glyphs next to each other. This should be reflected in the post table listing of these glyphs, which is the case as can be seen in the entries below:

```
<PostScriptName glyphRefID="759" NameString="notequal_clone" />
<PostScriptName glyphRefID="760" NameString="notequal_mirror" />
<PostScriptName glyphRefID="761" NameString="approxequal_clone" />
<PostScriptName glyphRefID="762" NameString="approxequal_mirror" />
<PostScriptName glyphRefID="763" NameString="integral_clone" />
<PostScriptName glyphRefID="764" NameString="integral_mirror" />
<PostScriptName glyphRefID="765" NameString="radical_clone" />
<PostScriptName glyphRefID="766" NameString="radical_mirror" />
<PostScriptName glyphRefID="767" NameString="summation_clone" />
<PostScriptName glyphRefID="768" NameString="summation_mirror" />
<PostScriptName glyphRefID="769" NameString="partialdiff_clone" />
<PostScriptName glyphRefID="770" NameString="partialdiff_mirror" />
```

If they are not in the right sequence, your font tool re-arranged the glyphs during generation of the font file. In the case of ***FontLab***, the setting that needs correcting is in the "Options" dialog box. Bring up the TrueType options panel and ensure "Use Unicode indexes as a base for TrueType encoding" is checked. With ***FontLab*** 3.x, to make it do the right thing, you may also need to delete the glyphs from the font, save, paste them back in and re-save, and then regenerate the TrueType file.

Making the XML edit using a "Delta" file

The biggest difficulty in font editing is to avoid errors due to the repetitive nature of the work. This is never truer than when scrolling through miles of XML listings to locate scattered entries and make small but critical changes. Using an intermediate text file to hold the editing changes helps this process greatly.

One useful technique is to locate the entries in the original dump file, *cut* them out and paste them together into a new "delta" text file. In here you can add working notes, you can group the entries in useful ways for checking, and also have a "before" and "after" set. Once the edits are done, you then paste the modified block back into the source file you cut it from (or a copy of it—leaving the original so you can repeat the process and/or check you made the right cuts). This out-of-sequence block pasting works because the order of entries in a dump file is not critical, as ***ftxdumperfuser*** sorts entries before writing them to the font file.

Benefits of this method are: (i) searching the dump file is reduced to one time only which cuts down user fatigue/boredom; (ii) the entries can be grouped next to each other so related edits can be made in sight of each other; (iii) there is a documented record of the edits made which is invaluable for Quality Assurance and version history tracking. In short, the method is supportive and self-documenting, resulting in faster, more accurate work, especially for large files. Let's look at a live example:

Create a new text file `31_PostDeltas.xml`, cut the mirror entries and original glyph entries out of the post table dump file `30_AppleSimple.post.xml`, and then save the dump file. You will find the new glyphs are all bunched together, as shown in the block above; however, the original glyphs are scattered in the file so you will have to locate each by searching on their names (notequal, approxequal, etc.). Once in the delta file, you can group them together thus:

```
The before entries

       <PostScriptName glyphRefID="350" NameString="notequal" />
       <PostScriptName glyphRefID="759" NameString="notequal_clone" />
       <PostScriptName glyphRefID="760" NameString="notequal_mirror" />

       <PostScriptName glyphRefID="368" NameString="approxequal" />
       <PostScriptName glyphRefID="761" NameString="approxequal_clone" />
       <PostScriptName glyphRefID="762" NameString="approxequal_mirror" />

       <PostScriptName glyphRefID="353" NameString="integral" />
       <PostScriptName glyphRefID="763" NameString="integral_clone" />
       <PostScriptName glyphRefID="764" NameString="integral_mirror" />

       <PostScriptName glyphRefID="364" NameString="radical" />
       <PostScriptName glyphRefID="765" NameString="radical_clone" />
       <PostScriptName glyphRefID="766" NameString="radical_mirror" />

       <PostScriptName glyphRefID="357" NameString="summation" />
       <PostScriptName glyphRefID="767" NameString="summation_clone" />
       <PostScriptName glyphRefID="768" NameString="summation_mirror" />

       <PostScriptName glyphRefID="356" NameString="partialdiff" />
```

```
<PostScriptName glyphRefID="769" NameString="partialdiff_clone" />
<PostScriptName glyphRefID="770" NameString="partialdiff_mirror" />
```

The editing and verification inspection now become trivial: First rename the original glyphs by adding an ".unused" suffix to each. Then remove the "_clone" from the end of the six glyphs containing it. The Delta file is not fed into any tool so it can contain both the before and after entries. This is invaluable in tracing bugs later as the file can be far more quickly inspected than scrolling through the entire dump.

```
The after entries

<PostScriptName glyphRefID="350" NameString="notequal.unused" />
<PostScriptName glyphRefID="759" NameString="notequal" />
<PostScriptName glyphRefID="760" NameString="notequal_mirror" />

<PostScriptName glyphRefID="368" NameString="approxequal.unused" />
<PostScriptName glyphRefID="761" NameString="approxequal" />
<PostScriptName glyphRefID="762" NameString="approxequal_mirror" />

<PostScriptName glyphRefID="353" NameString="integral.unused" />
<PostScriptName glyphRefID="763" NameString="integral" />
<PostScriptName glyphRefID="764" NameString="integral_mirror" />

<PostScriptName glyphRefID="364" NameString="radical.unused" />
<PostScriptName glyphRefID="765" NameString="radical" />
<PostScriptName glyphRefID="766" NameString="radical_mirror" />

<PostScriptName glyphRefID="357" NameString="summation.unused" />
<PostScriptName glyphRefID="767" NameString="summation" />
<PostScriptName glyphRefID="768" NameString="summation_mirror" />

<PostScriptName glyphRefID="356" NameString="partialdiff.unused" />
<PostScriptName glyphRefID="769" NameString="partialdiff" />
<PostScriptName glyphRefID="770" NameString="partialdiff_mirror" />
```

Make a copy of the chopped dump file `30_AppleSimple.post.xml` and paste the modified entries into the listing at either the bottom or the top. White space formatting does not matter (it is all treated as a single blank in XML) and it helps to leave the blank lines in so you can see where you have been.

```
cp 30_AppleSimple.post.xml 32_AppleSimple.post.xml
```

Fuse it back into a forward copy of the font, then immediately dump again to make sure everything went OK:

```
/Developer/Tools/CpMac 22_AppleSimple.ttf 33_AppleSimple.ttf

ftxdumperfuser -t post -d 32_AppleSimple.post.xml 33_AppleSimple.ttf

ftxdumperfuser -t post -o 34_AppleSimple.post.xml 33_AppleSimple.ttf

ftxdiff -o 35_PostDiff.txt 33_AppleSimple.ttf 22_AppleSimple.ttf
```

If all went well, nothing but the 'post' table (and checksum) changed.

Editing the cmap table clone entries

We need to dump the cmap to edit the clone glyph entries. To help us find the entries in the cmap, we will add the glyph names and the Unicode names to the 'cmap' dump. We do this by using the **-p** and **-u** options when dumping the 'cmap'.

```
ftxdumperfuser -pu -t cmap -o 36_AppleSimple.cmap.xml
      33_AppleSimple.ttf
```

By the way, the dump (-pu) options can go before or after the table selector, so you could also execute the command this way:

```
ftxdumperfuser -t cmap -pu -o 36_AppleSimple.cmap.xml
      33_AppleSimple.ttf
```

What needs to happen here is that the cmap entries for the six math characters need to be changed to point to the newly named copies which are next to the mirror pairs. This is done by altering the glyph index value which is labeled GlyphRefID in the XML. See the six cmap XML entries below.

Editing by use of a delta file is again helpful here as we can double-check the glyph index values before and after editing. Create a new text file `40_CmapDeltas.txt` and cut the six cmap entries out of the dump into the delta file.

```
<map charValue="0x2202" charName="PARTIAL DIFFERENTIAL" glyphRefID="356"
      glyphName="partialdiff.unused"/>
<map charValue="0x2211" charName="N-ARY SUMMATION" glyphRefID="357"
      glyphName="summation.unused"/>
<map charValue="0x221A" charName="SQUARE ROOT" glyphRefID="364"
      glyphName="radical.unused"/>
<map charValue="0x222B" charName="INTEGRAL" glyphRefID="353"
      glyphName="integral.unused"/>
<map charValue="0x2248" charName="ALMOST EQUAL TO" glyphRefID="368"
      glyphName="approxequal.unused"/>
<map charValue="0x2260" charName="NOT EQUAL TO" glyphRefID="350"
      glyphName="notequal.unused"/>
```

Open the previous postname delta file `31_PostDeltas.xml` to see the handy list of all related glyphs. For convenience, copy this into the `40_CmapDeltas.txt` so you're working in the same window and then do the edit: find the appropriate glyph in the postname list by glyph name and look up the `glyphRefID` of the new clone glyph and insert that value in the `glyphRefID` of the cmap entry, replacing the old value. At the same time, delete or amend the glyph name as they are no longer synchronized. The entries should end up as below. You can see all the glyph IDs have shifted from the 300s to the 700s.

```
<map charValue="0x2202" charName="PARTIAL DIFFERENTIAL" glyphRefID="769" />
<map charValue="0x2211" charName="N-ARY SUMMATION" glyphRefID="767" />
<map charValue="0x221A" charName="SQUARE ROOT" glyphRefID="765" />
<map charValue="0x222B" charName="INTEGRAL" glyphRefID="763" />
<map charValue="0x2248" charName="ALMOST EQUAL TO" glyphRefID="761" />
<map charValue="0x2260" charName="NOT EQUAL TO" glyphRefID="759" />
```

Make a copy of the cmap dump file `36_AppleSimple.cmap.xml` (from which you removed the entries you're editing) and paste the modified entries back into the empty spot.

```
cp 36_AppleSimple.cmap.xml 41_AppleSimple.cmap.xml
```

Fuse it back into a forward copy of the font, then immediately dump again to make sure everything went OK:

```
/Developer/Tools/CpMac 33_AppleSimple.ttf 42_AppleSimple.ttf

ftxdumperfuser -t cmap -d 41_AppleSimple.cmap.xml 42_AppleSimple.ttf

ftxdumperfuser -t cmap -pu -o 43_AppleSimple.cmap.xml
      42_AppleSimple.ttf

ftxdiff -o 44_CmapDiff.txt 42_AppleSimple.ttf 33_AppleSimple.ttf
```

If all went well, nothing but the 'cmap' table (and checksum) changed. The diff file reports succinctly that the expected re-mappings have been made.

```
0x2202 ==> 356 ==> 769
0x2211 ==> 357 ==> 767
0x221A ==> 364 ==> 765
0x222B ==> 353 ==> 763
0x2248 ==> 368 ==> 761
0x2260 ==> 350 ==> 759
```

Using glyph names instead of glyphRefIDs to edit the cmap

Working with glyph index values (glyphRefIDs) is the direct way of editing the cmap as this is what is actually stored in the cmap, but there are a couple of notable disadvantages: (i) Glyph IDs vary with each font so there is nothing that can be re-used. (ii) Numbers are less recognizable than names so it is harder to check the file for correctness. In short, editing and checking are more work.

The alternative is to use glyph names. These can be human-readable and are independent of the physical glyph order in a font, so they can be re-used between different fonts. This is, in fact, the whole purpose of the 'post' table's existence—that it contains stable name identifiers for glyphs so that the user is freed from transient index numbers.

Let us re-work the cmap edit above using the names rather than the indexes. Examining the starting point again—shown below—we see the cmap entries point to the old glyphs in the 300 index range instead of the new glyphs in the 700 index range and that each of the old glyphs has the postname with the "`.unused`" suffix.

```
<map charValue="0x2202" charName="PARTIAL DIFFERENTIAL" glyphRefID="356"
      glyphName="partialdiff.unused"/>

<map charValue="0x2211" charName="N-ARY SUMMATION" glyphRefID="357"
      glyphName="summation.unused"/>

<map charValue="0x221A" charName="SQUARE ROOT" glyphRefID="364"
      glyphName="radical.unused"/>

<map charValue="0x222B" charName="INTEGRAL" glyphRefID="353"
      glyphName="integral.unused"/>

<map charValue="0x2248" charName="ALMOST EQUAL TO" glyphRefID="368"
      glyphName="approxequal.unused"/>

<map charValue="0x2260" charName="NOT EQUAL TO" glyphRefID="350"
      glyphName="notequal.unused"/>
```

Instead of altering the index values, where we have to go and look up each individual index value of the new glyphs, we will instead alter the glyph name attribute. This field is normally passive in the cmap XML (i.e., it is just there for information and is ignored by ***ftxdumperfuser***). However, ***ftxdumperfuser*** has an option (**`-G`**) that tells it to use glyph *names* instead of glyph *numbers* when fusing in a '`cmap`'.

So to alter the cmap with the `-G` option, we put in the glyph names of the glyph we want to be associated with each cmap entry and ignore the glyphRefID. In this case the name we need is the original name of each of the math glyphs that we have placed on the new clone glyphs. Therefore the edit is to simply delete the ".unused" suffix from each glyph name, as shown below:

```
<map charValue="0x2202" charName="PARTIAL DIFFERENTIAL" glyphRefID="356"
      glyphName="partialdiff"/>

<map charValue="0x2211" charName="N-ARY SUMMATION" glyphRefID="357"
      glyphName="summation"/>

<map charValue="0x221A" charName="SQUARE ROOT" glyphRefID="364"
      glyphName="radical"/>

<map charValue="0x222B" charName="INTEGRAL" glyphRefID="353"
      glyphName="integral"/>

<map charValue="0x2248" charName="ALMOST EQUAL TO" glyphRefID="368"
      glyphName="approxequal"/>

<map charValue="0x2260" charName="NOT EQUAL TO" glyphRefID="350"
      glyphName="notequal"/>
```

Notice that now the glyphRefIDs are out of sync as they still refer to the old glyphs. ***ftxdumperfuser*** will give us a warning for each entry when it finds that the name and

index don't match. This useful feedback can be ignored for these intentional changes. An example warning message is shown below:

```
##### WARNING:  glyph name ("partialdiff") does not correspond to glyph number
      356, as it should, but to glyph number 769
```

So as before, prepare the entries in a delta file; we've added the entries to the bottom of the previous cmap delta file `40_CmapDeltas.txt`. Then make a fresh copy of the cmap dump file `36_AppleSimple.cmap.xml` (with the entries in question removed) and paste the modified entries into the missing section. (You could also just edit the cmap dump directly.)

```
cp 36_AppleSimple.cmap.xml 45_AppleSimple.cmap.xml
```

Make a fresh forward copy of the font from before the previous cmap edit and then fuse the 'cmap' in using the `-G` option. This will read our updated postnames and assign the mappings correctly.

```
/Developer/Tools/CpMac 33_AppleSimple.ttf 46_AppleSimple.ttf

ftxdumperfuser -G -t cmap -d 45_AppleSimple.cmap.xml 46_AppleSimple.ttf
```

Again we then dump the cmap straight out again to make sure that everything went OK and double-check this with ***ftxdiff***.

```
ftxdumperfuser -t cmap -pu -o 47_AppleSimple.cmap.xml
      46_AppleSimple.ttf

ftxdiff -o 47_CmapDiff.txt 33_AppleSimple.ttf 46_AppleSimple.ttf
```

Again, the diff file succinctly reports that the same changes have been made by this alternate method:

```
0x2202 ==> 356 ==> 769
0x2211 ==> 357 ==> 767
0x221A ==> 364 ==> 765
0x222B ==> 353 ==> 763
0x2248 ==> 368 ==> 761
0x2260 ==> 350 ==> 759
```

Default glyph properties table generation

To complete Mac OS X Unicode support we need to add a 'prop' table to the font. This Apple-specific table supports advanced line-layout features such as bidirectional text.

The 'prop' table stores information about each glyph, such as its directionality, whether it mirrors (a.k.a. "Directional paired glyphs"), if it's a non-spacing mark, whether it's a hanging glyph for paragraph justification, and whether it has an attachment to the next glyph on the right. You can read the spec of the table on-line at http://developer.apple.com/fonts/TTRefMan/RM06/Chap6prop.html.

The '**prop**' table describes glyphs rather than characters. This is so it can contain information that is very specific to a glyph's spatial design, such as special justification requirements. However, many of a font's glyphs have properties that are generic to their glyph name or character mapping identity. This enables us to make a good start at a '**prop**' table for our design by looking up the properties of each glyph in the glyph name database. This contains default properties for all the Unicode characters plus many of Apple's un-encoded glyphs, such as the Arabic display form shown in the sample entries below. The complete list of Unicode mirrored characters is on-line at http://www.unicode.org/Public/UNIDATA/BidiMirroring.txt.

```
<characterDatabase date="2002.05.02" version="1.0">
<charData name=".notdef" vChar="0x0F0000" properties="11" flags="0xFFFE" />
<charData name=".null" vChar="0x000000" properties="0" flags="0xFFFE" />
<charData name="A" vChar="0x000041" properties="0" flags="0xFFFE" />
<charData name="AE" vChar="0x0000C6" properties="0" flags="0xFFFE" />
<charData name="AEacute" vChar="0x0001FC" properties="0" flags="0xFFFE" />
<charData name="partialdiff" vChar="0x002202" properties="11" flags="0xFFFE" />
<charData name="summation" vChar="0x002211" properties="11" flags="0xFFFE" />
<charData name="radical" vChar="0x00221A" properties="11" flags="0xFFFE" />
<charData name="integral" vChar="0x00222B" properties="11" flags="0xFFFE" />
<charData name="approxequal" vChar="0x002248" properties="11" flags="0xFFFE" />
<charData name="notequal" vChar="0x002260" properties="11" flags="0xFFFE" />
<charData name="kashidaautoarabic" vChar="0x0F05E9" properties="11" flags="0xFFFE" />
```

ftxanalyzer has a **-P** option that reads the Unicode character properties from this database and generates a 'prop' table for the font. Now that we have all the cmap entries we want, we can use ***ftxanalyzer*** to generate a default 'prop' table for the font:

```
ftxanalyzer -P 48_AppleSimple.prop.xml 46_AppleSimple.ttf
```

The XML entries generated by ftxanalyzer for the six new math glyphs and their mirrored counterparts are shown in the excerpt below:

```
<glyphProps glyphRefID="757" glyphName="ogonekcmb" />
<glyphProps glyphRefID="758" glyphName="soliduslongoverlaycmb" />
<glyphProps glyphRefID="759" glyphName="notequal"
      directionalityClass="ON" />
<glyphProps glyphRefID="760" glyphName="notequal_mirror"
      directionalityClass="ON" />
<glyphProps glyphRefID="761" glyphName="approxequal"
      directionalityClass="ON" />
<glyphProps glyphRefID="762" glyphName="approxequal_mirror"
```

```
      directionalityClass="ON" />
<glyphProps glyphRefID="763" glyphName="integral"
      directionalityClass="ON" />
<glyphProps glyphRefID="764" glyphName="integral_mirror"
      directionalityClass="ON" />
<glyphProps glyphRefID="765" glyphName="radical"
      directionalityClass="ON" />
<glyphProps glyphRefID="766" glyphName="radical_mirror"
      directionalityClass="ON" />
<glyphProps glyphRefID="767" glyphName="summation"
      directionalityClass="ON" />
<glyphProps glyphRefID="768" glyphName="summation_mirror"
      directionalityClass="ON" />
<glyphProps glyphRefID="769" glyphName="partialdiff"
      directionalityClass="ON" />
<glyphProps glyphRefID="770" glyphName="partialdiff_mirror"
      directionalityClass="ON" />
```

The present version of ***ftxanalyzer*** sets the directionality class but not the paired cross-references, so these have to be added to each mirrored pair by hand in the following way, as illustrated by the bracket pair:

```
<glyphProps glyphRefID="86" glyphName="bracketleft" mirrors="YES"
      mirrorGlyphRefID="88" mirrorGlyphName="bracketright"
      directionalityClass="ON" />

<glyphProps glyphRefID="88" glyphName="bracketright" mirrors="YES"
      mirrorGlyphRefID="86" mirrorGlyphName="bracketleft"
      directionalityClass="ON" />
```

Three attributes need to be added to the XML: "mirrors", "mirrorGlyphRefID" and "mirrorGlyphName". As can be seen, the two references have to point at each other e.g.

```
<glyphProps glyphRefID="759" glyphName="notequal"  mirrors="YES" mirrorGlyphRefID="760"
      mirrorGlyphName="notequal_mirror" directionalityClass="ON" />

<glyphProps glyphRefID="760" glyphName="notequal_mirror"   mirrors="YES"
      mirrorGlyphRefID="759" mirrorGlyphName="notequal"  directionalityClass="ON" />
```

Edit the full set of changes in a text file `49_propDeltas.txt`, update the prop file and then fuse the updated file into the font:

```
cp 48_AppleSimple.prop.xml 50_AppleSimple.prop.xml

/Developer/Tools/CpMac 46_AppleSimple.ttf 51_AppleSimple.ttf

ftxdumperfuser -t prop -d 50_AppleSimple.prop.xml 51_AppleSimple.ttf

ftxdiff -o 52_PropEditDiff.txt 51_AppleSimple.ttf 46_AppleSimple.ttf
```

Notice that ***ftxdiff*** now reports that the `'prop'` table has been added but doesn't parse it further as there was no prop table present before to compare it with.

Final Unicode cmap clean up

It's now time to examine and clean up the remainder of the Unicode 'cmap'. Let's start with a freshly dumped copy of the cmap:

```
ftxdumperfuser -t cmap -pu -o 53_AppleSimple.cmap.xml 51_AppleSimple.ttf
```

We've already taken care to update the cmap with entries for the new mirrored glyphs we added; and the add lists have also automatically added their own entries. What are the remaining kinds of problems we need to look for in the cmap?

What remains is to look for gaps and any deadwood (i.e. redundant or false entries). Gaps are hard to spot and require a systematic reconciliation against the glyph repertoire along with some reference document for the intended mapping. Let's discuss dead cmap entries first:

Think of a dead Unicode cmap entry (i.e. one which has an incorrect glyph) as "false advertising" to the system. This is something that is very undesirable because of the font fall-back mechanism in Mac OS X: if a character is not in the presently selected font, the system will sometimes search *all* Unicode cmaps of *all* installed fonts to find the needed Unicode character. Therefore, any spurious cmap entries in your font could get picked up and used by the system as a fall-back resulting in a bad glyph display. This defeats the whole purpose of the fall-back mechanism. If there were no spurious entries, the System would continue searching other cmaps and would eventually find the correct glyph or display the Last Resort font glyph rather than your `.notdef` box.

Some unwanted entries are easy to spot using the Unicode character database names. In particular, any characters labeled "Unassigned" or "Private Use Character". These name labels are included in the dump by use of the `-u` option. The present Unicode data set used in these tools is version 3.2.0. These are therefore the first entries you should examine for deadwood.

Unassigned characters

It is possible to have Unicode values in a cmap that are named "Unassigned". This is because some codes that were present in earlier versions of Unicode have now been removed. An example of this is shown in Apple Simple from `0xD800`–`0xDC4D`:

```
<map charValue="0xD800" charName="[Unassigned U+D800]" glyphRefID="605"
      glyphName="uniD800"/>
. . . down to . . .
<map charValue="0xDC4D" charName="[Unassigned U+DC4D]" glyphRefID="683"
      glyphName="uniDC4D"/>
```

These characters were for support of unpaired surrogates in the font. This is a relic from the pre-Mac OS X 10.1 days, when there was no direct support for Unicode characters above `U+FFFF`. (Unicode is divided into planes of 65,536 code points each; the planes above `U+FFFF` are called the *supplementary*.) These mappings can be deleted.

Different kinds of Private Use Area (PUA) Mappings

The PUA begins at U+E000 and ends at U+F8FF. This area can contain character mappings of varying status, some of which should be removed and others kept. Below are some examples that are in no way a complete description of what you can find in this block:

```
<map charValue="0xE600" charName="[Private Use Character U+E600]"
      glyphRefID="165" glyphName="uniE600"/>
```

(1) FontLab 3.x PUA scratch space 0xE000: The first part of the PUA is a 'public' private use area. This is because it is the default assignment area used by ***Fontlab 3.x*** for any glyphs without explicit Unicode entries. This version of the tool forced such assignments because it used the Unicode value as the primary internal glyph identifier. ***FontLab 4.x*** no longer does this. However, any font opened and re-generated with (say) ***FontLab 3.1.3*** on Mac OS will contain at least a few mappings of the un-encoded glyphs up in this range starting at the first available slot in the U+E000 block and incrementing from there (i.e. U+E001, U+E002, etc.). If you identify any such glyphs, they should be deleted from the cmap. In the case of Apple Simple, there are none of these and the U+E000 range is filled with mappings of the following kinds:

(2) Glyphs for scripts not yet encoded in Unicode. There are two examples of these in Apple Simple: a) **Pollard**, a script for some minority languages in southwest China, is mapped from U+E600–U+E66C plus supporting glyphs from U+E670–U+E6AF; b) **Shavian**, a phonetic script for English, is mapped from U+E700–U+E72F. The latter is from a popular registry of PUA mappings called ConScript, which you can view at http://www.evertype.com/standards/csur/. Scripts are continually being investigated and added to Unicode, so you should also check to see if your script is in the pipeline for Unicode inclusion. Check the status of scripts that are in the process of being encoded in Unicode at http://www.unicode.org/unicode/alloc/Pipeline.html. You will notice that Pollard and Shavian are in the pipeline. What stage have they reached and when will they need re-mapping?

(3) Glyphs for scripts that have moved into Unicode. In this case, the **Deseret Alphabet** used to be unencoded and was mapped in the PUA from U+E830–U+E885. However, it was added in Unicode 3.1 to the supplementary planes, so this old ConScript registry PUA mapping needs moving. The re-mapping process is outlined below.

(4) Glyphs from Corporate PUAs. Various computer companies declare their use of the PUA for data interchange. This is a semi-formal convention to try to avoid mapping collisions. In this case, Apple Simple contains the Apple logo code point (U+F8FF) from the Apple Corporate PUA mapping. Apple's full PUA use is documented at http://www.unicode.org/Public/MAPPINGS/VENDORS/APPLE/CORPCHAR.TXT. You can also view other companies' mappings at http://www.unicode.org/Public/MAPPINGS/VENDORS/. There are also Asian font code pages in the PUA for gaiji characters and mobile phone text symbols (e.g. DoCoMo). The PUA can be a pretty busy place, so you don't want to leave redundant cmap entries in there: they could get picked up and interpreted as any of these other mappings.

Reconciliation of normal Unicode cmap entries

Having removed/corrected any unassigned characters and accounted for all the entries in the PUA, any other cmap entries by definition must be valid Unicode code points. That means they have a defined appearance and properties that you can check against the published Unicode standard. This is the slow part, as each individual cmap entry needs to be checked. You can do this automatically against the glyph name if you have confidence in the names. Failing that, you will need to eyeball the actual glyphs in the font to check their identity and the correctness of the glyph names at the same time as you check the mappings.

One approach that works is to make a master reference document, spreadsheet, or database that contains the names and mappings of your font design to help you in this process. This is commonly called a "Glyph Registry". It is best built as part of the planning process before the font is created because it can then act as the master source for generating the glyph names and cmap entries. This reduces production work, reduces errors in creating identifiers and facilitates checking at all stages.

Which Apple Simple Unicode cmap entries need editing?

Cut out the unassigned mappings (U+D800–U+DC4D), the Pollard display variants (U+E670–U+E6AF) and make a decision about Deseret (U+E830–U+E885): either cut and re-generate from scratch or modify the existing entries. A color-coded cmap showing the various blocks can be viewed in 54_CmapCheck.pdf.

Re-mapping the Deseret script block to the supplementary planes

Re-mapping the Deseret cmap entries from their current code assignments must be done by editing the ***ftxdumperfuser*** cmap dump file. This is because the Deseret code points, which start at U+10400, are outside the normal Unicode range, being in the supplementary planes, and most font tools don't support these mappings yet.

Deseret has two cases: uppercase, which is at U+E830–U+E855 in the Apple Simple cmap and now moves to U+10400–U+10425, and lower-case, which is at U+E860–U+E885 in the Apple Simple cmap and now moves to U+10428–U+1044D. Fortunately, the letters are in the same order, so we just have to change "0xE830" to "0x10400", and so on.

The two reference documents you need to have are the Deseret code block chart from the Unicode web site at http://www.unicode.org/charts/PDF/U10400.pdf and the Apple Simple glyph palette printout in the file 29_AppleSimple.pdf, which is in the Lesson Two folder. Three of the Apple Simple cmap entries are shown below:

```
<map charValue="0xE830" charName="[Private Use Character U+E830]"
      glyphRefID="396" glyphName="Longideseret"/>
<map charValue="0xE831" charName="[Private Use Character U+E831]"
      glyphRefID="397" glyphName="Longedeseret"/>
<map charValue="0xE832" charName="[Private Use Character U+E832]"
      glyphRefID="398" glyphName="Longadeseret"/>
```

The manual editing needed is to remove the `charName` attribute as this is passive data (this is optional since ***ftxdumperfuser*** will ignore this data unless you ask it not to), and to then edit the `charValue` entries to map to the new code points. The entries above would therefore be modified as shown below:

```
<map charValue="0x10400"  glyphRefID="396" glyphName="Longideseret"/>
<map charValue="0x10401"  glyphRefID="397" glyphName="Longedeseret"/>
<map charValue="0x10402"  glyphRefID="398" glyphName="Longadeseret"/>
```

Re-mappings such as this are entirely determined and are best done automatically using a macro, script, spreadsheet or database to avoid introducing human editing errors. The choice of automation method depends on your particular computing resources.

This exercise is left to the reader: one approach is to save a copy of the cmap with the Deseret entries removed; paste those entries into a delta file to operate on; perform the re-mapping by your chosen method; paste the modified entries back into a copy of the cmap file, then fuse back into the font and check that all is as expected.

Mac Roman cmap check

Finally, the MacRoman cmap should be checked. The ***ftxanalyzer*** `-M` option made sure that the Unicode cmap contained all the characters in MacRoman, but it did not check the actual MacRoman cmap itself. The first check is to count the entries. One way is by copying the entries into a fresh document window and counting the lines with a get info or line number option (some editors will also show you the number of lines in the current selection). You will find that Apple Simple's MacRoman cmap contains only 99 out of the 256 characters in MacRoman. This is probably because the cmap was generated very early on in the life of the font when it only had a partial repertoire. Since then, the tools have not automatically updated the cmap. One way to fix it is to delete the MacRoman cmap subtable entirely, which will trigger some tools to re-generate it afresh as part of a "Create Macintosh font" option.

Another sure way to fix it is to edit the XML source file directly, which will involve looking up the mappings in a reference document. There are several starting places you can choose for this, depending on your programming skills and on which identifiers you have done the most validation work.

(1) Manual editing: This is the simplest but most time-consuming method, as you must look up and edit everything by hand and the time spent is not leveraged in any way. As we are not doing any automatic scripting or formatting, it is best to start with a pre-existing chunk of XML for the MacRoman cmap and to modify that rather than have to create all the XML tags by hand. We recommend you dump a complete MacRoman cmap from a trusted system font such as Lucida Grande or Times. A copy of the Lucida MacRoman cmap entries is in the text file: `55_LucidaMacRoman.cmap`. An excerpt of this is show below:

```
<map charValue="0x0026" glyphRefID="9" glyphName="ampersand"/>
<map charValue="0x0027" glyphRefID="10" glyphName="quotesingle"/>
<map charValue="0x0028" glyphRefID="11" glyphName="parenleft"/>
<map charValue="0x0029" glyphRefID="12" glyphName="parenright"/>
```

The manual task is for every entry, to look up the glyph in the Apple Simple glyph palette printout in the file `29_AppleSimple.pdf`. Use the glyph name and the shape to look up the correct glyph in the palette. When you have found the glyph, note the index number and change the glyphRefID in the cmap to the index number of that glyph.e.g. the excerpts above will be changed as below:

```
<map charValue="0x0026" glyphRefID="33" glyphName="ampersand"/>
<map charValue="0x0027" glyphRefID="34" glyphName="quotesingle"/>
<map charValue="0x0028" glyphRefID="35" glyphName="parenleft"/>
<map charValue="0x0029" glyphRefID="36" glyphName="parenright"/>
```

(2) Automation using Glyph names as primary glyph identifiers: You can either parse an XML MacRoman cmap dump to extract the glyph names (e.g. `55_LucidaMacRoman.cmap`) or start with the tab delimited table provided in the file `56_MacRomanPostnames.txt` an excerpt of which is shown below:

```
HEX   GLYPH NAME
0026  ampersand
0027  quotesingle
0028  parenleft
0029  parenright
```

The second data source you need is the post name dump. Either make a fresh dump or use the earlier post dump `34_AppleSimple.post.xml`, which contains the names of the first 874 glyphs just prior to the last big diacritic add list addition. An except is shown below:

```
<PostScriptName glyphRefID="33" NameString="ampersand" />
<PostScriptName glyphRefID="34" NameString="quotesingle" />
<PostScriptName glyphRefID="35" NameString="parenleft" />
<PostScriptName glyphRefID="36" NameString="parenright" />
```

You now need to combine the two by looking up the postnames between the files using whatever programming resources you have available, such as `grep`, `perl`, shell scripts, WP macros, programming routines, spreadsheet macros, database scripts, etc.

Note that the string match has to be exact as the names are case sensitive. (i.e. "eacute" and "Eacute" are different glyphs). If in your programming environment you cannot do a hard string comparison, you will have to convert the postnames into their hex representation before doing the lookup. e.g. Note the difference in upper and lower case E at the start of the two strings below (`E = 0x45; e = 0x65`)

```
Ecircumflex → 4563697263756D666C6578
ecircumflex → 6563697263756D666C6578
```

Once you have done the look-up and created the three columns, then tag them with the XML as shown below and you're done.

```
<map charValue="0x0026" glyphRefID="33" glyphName="ampersand"/>
<map charValue="0x0027" glyphRefID="34" glyphName="quotesingle"/>
<map charValue="0x0028" glyphRefID="35" glyphName="parenleft"/>
<map charValue="0x0029" glyphRefID="36" glyphName="parenright"/>
```

Note that if your postnames match the ones in the font you're using as a model, the remapping step is unnecessary; you can tell ***ftxdumperfuser*** to favor the names rather than the IDs when fusing.

(3) Automation using Unicodes as primary glyph identifiers: If you place your trust in the Unicode cmap more than the glyph names, then the reference document you need to generate the mapping is the official Apple MacRoman mapping document, a copy of which is in enclosed in the file named `57_MacRomanUnicodeMapping.TXT`.

```
MROM      UNICODE      # UNICODE CHARNAME
0x26      0x0026       # AMPERSAND
0x27      0x0027       # APOSTROPHE
0x28      0x0028       # LEFT PARENTHESIS
0x29      0x0029       # RIGHT PARENTHESIS
```

Note that although this is a long-standing code page, it is still always good to check for any changes in the latest published version, which is on-line at the Unicode web site `http://www.unicode.org/Public/MAPPINGS/VENDORS/APPLE/ROMAN.TXT`. The most recent change was the Euro glyph mapping added in 1998.

The second data source you need is the Unicode cmap dump, an excerpt of which is shown below where it is dumped without any glyphames:

```
                Unicode
<map charValue="0x0026" glyphRefID="33" />
<map charValue="0x0027" glyphRefID="34" />
<map charValue="0x0028" glyphRefID="35" />
<map charValue="0x0029" glyphRefID="36" />
```

The programming task here is to extract the values out of the two columns in each of the two references tables and combine the `glyphRefID`'s with the MacRoman mappings by matching on the Unicode value. One advantage of a Unicode look-up is that no hard string comparison is necessary as the strings are already in hex.

Once you have done the look-up and created the two columns, then tag them with the XML as shown below and you're done. Note that the MacRoman codepoints mirror the Unicode values for the low ASCII, but that they differ in the high bit values; so although the example entries below look identical to the Unicode cmap, the overall table is significantly different.

```
                MacRoman
<map charValue="0x0026" glyphRefID="33" />
```

```
<map charValue="0x0027" glyphRefID="34" />
<map charValue="0x0028" glyphRefID="35" />
<map charValue="0x0029" glyphRefID="36" />
```

If you have the data available, you can add the glyph names as this helps human double-checking. e.g. as the Lucida Grande MacRoman cmap excerpt below shows:

```
<map charValue="0x00DB" glyphRefID="917" glyphName="Euro"/>
<map charValue="0x00DC" glyphRefID="190" glyphName="guilsinglleft"/>
<map charValue="0x00DD" glyphRefID="191" glyphName="guilsinglright"/>
<map charValue="0x00DE" glyphRefID="1278" glyphName="fi"/>
<map charValue="0x00DF" glyphRefID="255" glyphName="fl"/>
```

The glyph names can also be added by fusing your new cmap, then dumping it out again with the –p option.

Detecting Gaps in the glyph repertoire

As mentioned at the beginning of the discussion of final Unicode cmap clean up above, the only way to detect gaps in the glyph repertoire is by comparing against reference data outside of the font. The MacRoman cmap check is an example of this. Were all the Mac Roman glyphs present in the font?

For the wider glyph repertoire beyond MacRoman, there is no handy reference document apart from Unicode unless you as the designer create one. We recommend this as part of the font planning and updating process. It is especially important if your design includes unencoded display variants. Such a text file, spreadsheet, or database of the expected glyphs, chosen glyph names, and encodings is invaluable in the final checking of the font.

Final repertoire clean up; eyeballing the glyph palette

Now that we have become more familiar with the structure of the font through working with the cmaps, it's good to go back over the physical glyph palette and see if anything can be cleaned up there. Sure enough, there are some dead glyphs:

(i) FontLab 3.x control glyphs. As well as adding Unicode cmap entries for all glyphs, FontLab 3.x also creates certain control block glyphs. You can prevent this by use of the latest 4.x version of FontLab or by editing of the FontLab text data mapping files. In Apple Simple glyph IDs 22-27 are examples of these (`controlNULL, controlBS, controlHT, controlLF, controlCR, controlGS`). These are normally mapped to `.null` and `nonmarkingreturn`. The only reasons to have separate glyphs for these characters would be if you wished to attach different properties than those of the .null or nonmarkingreturn or if you wanted to include these characters in unique morphing actions. These are very unlikely scenarios so these glyphs can be removed in most cases.

(ii) Surrogate glyphs: These are glyph IDs 605–696, which were for surrogate support in earlier versions of Mac OS X. We just cut the Unicode cmap entries for these unassigned characters so we might as well also remove the glyphs from the font.

Any other glyphs in the repertoire that are not in a cmap may still be live through being used by a shaping action such as a ligature. To know if a glyph is truly unused, it is necessary to check all the cmaps and all the MIF files.

Glyph Registry and Assignment Report

If you are maintaining a master font design document, it should include the uses of each glyph in shaping rules as well as the cmap entries. This glyph registry and accompanying "assignment report" annotations will then highlight any unused glyphs as they are simply the ones with no assignments.

The report will also highlight any mismatches between MacRoman and Unicode cmaps. For example, here are two live entries from a font that highlight a mix-up in the assignments of two identical glyphs: The MacRoman encoding should use glyph 257 and the ligature action should be on the combining form—not the ordinary form.

```
glyph name: macron
glyphRefID: 257
MacRoman:   -
Unicode:    0x00AF MACRON
Morph:      Ligature (default ON)

glyph name: macron.cmb
glyphRefID: 216
MacRoman:   0x00F8 macron
Unicode:    0x02C9 MODIFIER LETTER MACRON
Morph:      -
```

Glyph deletion and its consequences

Note that if we cut the dead glyphs, then the glyph indexes will all change. Consequently the XML dump files which contain glyphRefIDs will be out-of-sync with the font. To avoid this, it is recommended that the glyph repertoire be stabilized as much as possible first. Unexpected changes do occur, though, so we need ways to cope.

Deletion actions vary depending on the tool you use. We recommend ***FontLab 4.5*** for glyph deletion because its internal representation of glyphs as objects preserves the glyph name and Unicode attributes of each glyph independent of index order.

Whenever you do a glyph deletion, immediately do a dump of the post table, cmap table, and the glyph palette and check that the identifiers have not been scrambled in a domino shift. If they have, you will need to apply a re-mapping process on your XML text sources before you can continue.

Because of the vulnerability of the glyph indexes to deletions in the glyph palette, use glyph names in the text sources wherever possible. They will work as long as the post table is synchronized with the glyphs. This is especially true of shaping behavior rules, described in the next Lesson.

Because glyph deletion is so disruptive and the way you cope with it will depend on the support tools you have available, we will not cut the dead glyphs from Apple Simple in this Tutorial. It is left as an exercise for the reader to set up tracking documentation and re-mapping processes for supporting a major glyph shuffle.

There is also the glyph clean-up work to be done on the diacritic compositions made by the Add lists. Are there any other glyph corrections you have identified?

Lesson Four: Metamorphosis Input Files (MIFs)

Apple Advanced Typography (AAT) enables you, the font designer, to make glyphs behave in interesting and useful ways within text. Full support of AAT is available to applications in Mac OS X, and the number taking advantage of it is increasing all the time.

The heart of AAT is the 'morx' table in a font, which contains the "smarts" that AAT uses to do things like generate ligatures, rearrange glyphs, and so on. In this tutorial, we'll be adding AAT support to our font.

Historical note: Mac OS X uses 'morx' tables. Earlier systems used 'mort' tables. There is no difference in their function except that the 'morx' tables can support fonts with larger numbers of glyphs. 'morx' actually means '**mor**t **ex**tended' i.e. extended mort tables.

'morx' tables are added to fonts using ***ftxenhancer***. The file format used is referred to as a "Metamorphosis Input File", "Morph Input File" or "MIF." We'll be generating a MIF for our font that contains the features we want.

Generating a default MIF file for decomposed Unicode support

We again start with ***ftxanalyzer,*** which can generate a default MIF for a font through use of its -m option. The default MIF contains the basic shaping behaviors, the majority of which are for decomposed Unicode support i.e. the MIF will contain the list of precomposed Unicode characters and how they break down into composing pieces. Navigate to the Lesson Four folder and type:

```
ftxanalyzer -m 58_AppleSimple.mif 51_AppleSimple.ttf
```

Look through the resulting MIF file, 58_AppleSimple.mif, with a text editor. The MIF format is described in the Font Tool Suite document. In the MIF format, lines beginning with "//" are comments and lines consisting of only dashes are also ignored and can be used as section dividers.

The file is split up into a series of sub-tables, each with their own commented titles and header settings. A sub-table is the main unit of organization within the MIF and the resulting 'morx' table. The start of the file with the beginning of the first sub-table is shown below:

```
//=====================================================
//  Auto-generated UniMIF data
//    Decomposition length == 3
//    Decompsotion type == Full
//=====================================================

Type          LigatureList
Name          NULL
Namecode      27
Setting       NULL
Settingcode   0
Default       yes
Orientation   HV
Forward       yes
Exclusive     no

List

    uni01E0   A dotaccentcmb macroncmb // (#7) GID   830 == U+0041 + U+0307 + U+0304
    uni01DE   A dieresiscmb macroncmb  // (#9) GID   828 == U+0041 + U+0308 + U+0304
    uni01FA   A ringcmb acutecmb       // (#11) GID   849 == U+0041 + U+030A + U+0301
    uni022C   O tildecmb macroncmb     // (#63) GID   867 == U+004F + U+0303 + U+0304
    uni01EC   O macroncmb ogonekcmb   // (#65) GID   840 == U+004F + U+0304 + U+0328
    uni0230   O dotaccentcmb macroncmb //   (#68) GID   871 == U+004F + U+0307 + U+0304
```

The file is labeled "UniMIF" data, because it's a "Unicode MIF." The UniMIF entries handle Unicode composition and ensure that no matter whether an application uses precomposed Unicode or composing Unicode with your font, the text is rendered correctly.

In the UniMIF sub-table headers, the "Name" and "Setting" values are consistently NULL. This is a flag to ***ftxenhancer*** which tells it that these features should always be on and never off—with the side effect that there should be no switch visible in the user interface to turn them off. It is actually still possible for the user to deactivate them by turning off all metamorphosis actions, but we want to discourage this as much as possible.

Adding fi and fl ligature actions with ZeroWidthJoiner support

We want to add a couple of features ourselves. One is quite common, namely support for the fi and fl ligatures. Make a forward copy of the MIF `59_AppleSimple.mif` and add the following text at the end. A copy of the text is in the file `60_fiflLigSubtables.mif`.

```
cp 58_AppleSimple.mif 59_AppleSimple.mif
```

```
-----------------------------------------------------------------
//  Normal support for fi and fl
-----------------------------------------------------------------
Type  LigatureList
Name  Ligatures
```

```
Namecode    1
Setting     Common Ligatures
Settingcode 0
Default     yes
Orientation HV
Forward     yes
Exclusive   no

List
      fi  f i
      fl  f l

---------------------------------------------------------------------
//  Support for fi and fl using zerowidthjoiner
---------------------------------------------------------------------
Type        LigatureList
Name        NULL
Namecode    27
Setting     NULL
Settingcode 2
Default     yes
Orientation HV
Forward     yes
Exclusive   no

List
      fi f zerowidthjoiner i
      fl f zerowidthjoiner l
```

Why two subtables? Unicode 3.1 and later recommends that any ligature that is available in your font be automatically formed whenever the U+200D ZERO WIDTH JOINER (ZWJ) character is between its pieces, even if ligatures are otherwise turned off. (See the online copies of the Unicode Standard, versions 3.1 and 3.2, for more information.) In our case, we have only the two ligatures, fi and fl. We therefore have one version automatically formed whenever f and i or f and l are next to each other, unless the user has turned ligatures off. We also have the version that requires U+200D and is always on.

Notice that we don't have a ZWJ version of the ligature glyphs for Unicode composition. That's because, although we used a "ligature" feature to implement them, they're not actually ligatures, typographically speaking.

Sub-table sequence dependencies in the MIF file

You can control the order in which various actions take place by the order in which you put the sub-tables in the MIF. We want the UniMIF stuff to happen all the time, regardless, and before anything else happens. This is important for any other actions you define in the MIF, as prior actions will change the glyphs in the display buffer. In this case, the benefit of placing the UniMIF actions first is that any other shaping rules can be defined to work on the precomposed glyphs only rather than both precomposed and decomposed glyph sequences. Therefore, the UniMIF sub-tables are placed first in the MIF file.

Sequence of execution within a Ligature sub-table

All sub-tables in the MIF end up as state tables in the 'morx' table. This includes the simplified ligature list format shown here. A result of this is that the sequence of execution of the ligature actions can be different from the order of the ligature actions in the original MIF file. The sequence is predictable, however, so the designer can structure the MIF to take this into account:

(1) The actions are sorted first by length of the input string, with the longest patterns being executed first. For example, if you have a ligature for e+s+s it will fire before a ligature for e+s. Matching the longest context first is sensible default behavior.

(2) Actions of the same pattern length are sorted by glyph index in ascending order.

This order is reflected in the structure of the UniMIF data. The patterns of length 3 are placed in a sub-table before patterns of length 2. Within each sub-table, the actions are listed in ascending glyph index order.

When this execution order is problematic, the solution is to split the conflicting actions out into their own separate sub-table and place that sub-table either before or after the one they came from in order to override the execution sequence.

Adding Pollard shaping support

The font glyph repertoire supports Pollard, which requires some shaping behaviors to display correctly. Copy the contents of the `61_Pollard.mif` file and paste it at the end of the MIF file `59_AppleSimple.mif`. This MIF makes sure that Pollard finals combined with tone marks properly compose themselves. The start of the `61_Pollard.mif` file is shown below:

```
Type  LigatureList
Name  Ligatures
Namecode    1
Setting     Pollard tones
Settingcode 4
```

```
Default     yes
Orientation HV
Forward     yes
Exclusive   no

List
      uniE630     uniE630     uniE663
      uniE631     uniE631     uniE663
      uniE632     uniE632     uniE663
      uniE633     uniE633     uniE663
      uniE634     uniE634     uniE663
      uniE635     uniE635     uniE663
      uniE636     uniE636     uniE663
```

Notice, by the way, that since we're using glyph names for our Pollard glyphs, we don't need to worry about the actual glyph indices for the glyphs in question. The MIF can refer to glyphs by their glyph names, or by the Unicode code point they refer to, or by their glyph index. Of these, the Unicode code point and glyph names are the most flexible. Using them means you can cut and paste pieces between MIFs for different fonts, and they'll still work correctly.

Adding long-s typographic feature support

Finally, as our font has a long-s character in it, let's set things up so that it can be used in the middle of a word. Copy the contents of the file `62_LongS.mif` and add it to the end the MIF file `59_AppleSimple.mif`. The text of this file is shown below:

```
-----------------------------------------------------------------
//  Turn medial s into long s
-----------------------------------------------------------------
Type              Contextual
Name              Smart Swashes
Namecode    8
Setting           Medial Long-s
Settingcode 8
Default           no
Orientation H
Forward           yes
Exclusive   no

Ess         s
Lower       a b c d e f g h i j k l m n o p q r t u v w x y z

                  EOT     OOB     DEL     EOL     Ess     Lower
StartText   1     1       1       1       2       1
StartLine   1     1       1       1       2       1
SawS              1       1       1       1       3       4
SawSS             1       1       1       1       3       4

    GoTo          Mark?   Advance?        SubstMark       SubstCurrent
1   StartText     no      yes             none            none
2   SawS          yes     yes             none            none
3   SawSS         yes     yes             ToLongS         none
4   StartText     no      yes             ToLongS         none

ToLongS
    s       slong
```

Note a couple of things about this MIF: For one, it's not on by default as we wouldn't want that. Instead, it has to be activated by the user in the Typographic features palette of an application. The second point is that this sub-table uses a state table rather than a ligature list. State tables are powerful but complex. Again, the ability to re-use MIF definitions containing glyph names comes to our rescue. We wouldn't want to have to re-write this MIF more than once.

MIF state table writing can sometimes be a bit tricky. We plan to address this in future releases of the tools introducing friendlier input formats. Meanwhile, remember to use the -v option if something goes wrong; this will usually help figure out which line of the MIF was bad and make diagnosing problems *much* easier.

Compiling the MIF into the font

Now we can add the MIF. Make a forward copy of the font; run ***ftxenhancer*** with the -m option on it and then check the changes with ***ftxdiff***:

```
/Developer/Tools/CpMac 51_AppleSimple.ttf 63_AppleSimple.ttf

ftxenhancer -m 59_AppleSimple.mif 63_AppleSimple.ttf

ftxdiff -o 64_MIFDiff.txt 63_AppleSimple.ttf 51_AppleSimple.ttf
```

A working copy of the MIF is stored in `66_AppleSimple.mif` for reference. Upon successful fusing, we should find that everything is the same in the diff report, except that a '**`morx`**' table and a '**`feat`**' table have been added along with a set of new interface strings in the '**`name`**' table.

Font installation checks

So let's install the font and see what happens. First check that we don't already have a copy installed somewhere by using:

```
ftxinstalledfonts -fl | grep Simple
```

ftxinstalledfonts lists information about the fonts installed on your system. In this case, using the `-f` and `-l` options, we want to know the full names of fonts and the location where they're installed. The rest of the line passes the output of ***ftxinstalledfonts*** through a Unix utility called `grep` that looks for "regular expressions." In this case, we want it to only output lines passed into it that contain the word "Simple." If Apple Simple is not installed there will be no output. If "Apple Simple Thin" is installed already, we will get back output something like this:

```
14 8383 Apple Simple Thin /Users/tseng/Library/Fonts/Apple Simple
```

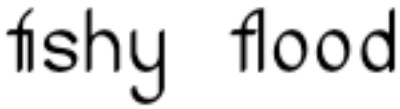
The "`14`" tells us that the font is fourteenth in the list of installed fonts. "`8383`" is the system's internal font ID for this particular system and boot session. "`Apple Simple Thin`" is the font's name, and "`/Users/tseng/Library/Fonts/Apple Simple`" is the place where it's installed—in this case, the user's home directory.

Assuming there isn't a conflict, install the font by dropping a copy of `63_AppleSimple.ttf` in your `~/Library/Fonts/` folder. To do this in the Finder, double-click on the OSX system volume to create a new window; click on the home icon and open the Library folder you see in this window. Then drop the file onto the "`Fonts`" folder icon within Library. A backup working copy of the font is also in the file `67_AppleSimple.ttf`.

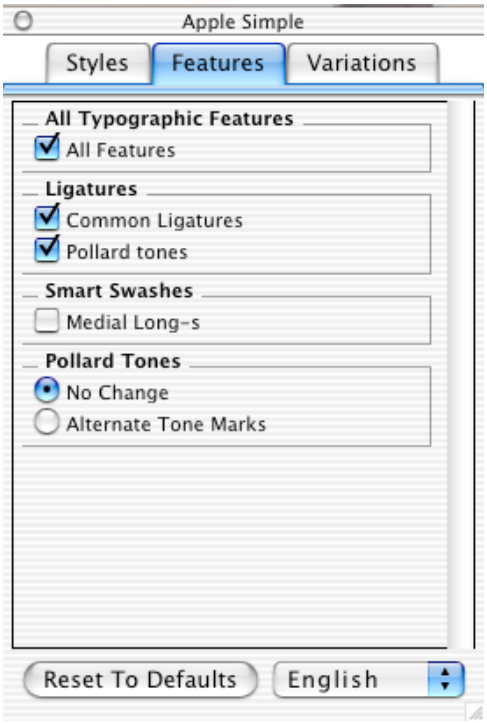
Application testing of the typographic features

Once the font is installed, two applications that come with Mac OS X can be used to test it. One is ***TextEdit***, the standard text editor, and the other is ***WorldText***, which is only installed if you've installed the Developer Tools. ***TextEdit*** only supports ligatures, but ***WorldText*** supports all of AAT. Since our font uses default ligatures for everything except the medial long-s, all features will work in ***TextEdit*** except for the medial long-s.

To test the entire set of features in a font, we recommend using WorldText, which has a "Typographic Features" palette. This enables you to control each feature individually to check its behavior, and also to turn off font substitution to make sure that it's your font that you're seeing. Locate ***WorldText*** in `/Developer/Applications/Extras` and launch it. Switch the font to Apple Simple, and type something like "fishy flood". You should see something like the following with both the fi and fl ligatures being used:



Make sure this really *is* Apple Simple by unchecking **Font Substitution** in the **Layout** menu. Select your text and bring up the typography palette from the **Window** menu (**Command-Y**). You should see something like the screen-shot below, with three custom features listed: Ligatures; Smart Swashes; and Pollard Tones.



Testing Zero Width Joiner support

Turn off common ligatures using the typographic features palette. The fi and fi ligatures should break apart. Move the insertion point between the f and the i, and insert U+200D ZERO WIDTH JOINER. The easiest way to do this is to use the Unicode Hex Input keyboard, which you can activate using the International preferences panel in System preferences, and then select using the input menu (which appears to the right of the application's menus).

To insert U+200D, select "Unicode Hex Input" from the input menu. While holding down the option key, type the four keystrokes 2 0 0 d. That will insert U+200D in the document—and the fi ligature reforms although ligatures are turned off.

Test of medial long-s form

Select all the text, and turn on medial long-s. You should see something like the "fishy flood" illustrated below:

fiſhy flood

Test of Shavian cmap

Finally, we can test Shavian and Pollard. For Shavian, just try letters at random starting at U+E700, using the Unicode Hex Input keyboard. You will get text looking like that shown below:

Test of Pollard shaping

Now insert the sequence <U+E600 U+E630 U+E660>. With features on (and the Pollard tones feature in particular), you should see something that looks like a Y with a macron, as illustrated below. This isn't an actual Y, and it isn't an actual macron, and the fact that the non-macron isn't exactly over the center of the non-Y is OK. That's what Pollard looks like.

If you now turn features off you should see the same sequence expanded into separate spacing glyphs with the Y being followed by the macron and two dotted circles, one with an F and one with an I, as shown below:

This is because of the ligature formation we put in the MIF. The Pollard initials, finals, and tones are combining correctly as the writing system requires.

What happens if you turn the Pollard Tones feature on and vary the last character from U+E660 to U+E661 and on through U+E664?

A WorldText document containing these demonstration strings is in the file 69_AppleSimpleDemo.wtx.

Creating Kerning and Justification input files

In addition to MIF files, you can also define text input files for Kerning (KIF) and Justification (JIF). These formats are described in the Font Tool Suite document.