# Unicode Symbols

**Abstract**
The *Unicode* standard includes a number of signs, symbols, dingbats, bullets, arrows, graphical elements, and other miscellaneous glyphs. Prompted by finding a font dedicated to many such *Unicode* symbols on MacOSX systems, this magazine documents some ways of enabling these symbols on your own system.

## Introduction

The UNICODE standard is dedicated to creating a universal character set for all of the languages on earth. Signs and symbols are often important components of and aids to printed communication. Appropriately enough, UNICODE dedicates a number of blocks to symbols, arrows, block elements, and geometric shapes that can be useful in some documents.

CONTEXT offers integrated support for symbols. As such, all that's necessary for CONTEXT support for UNICODE symbols is a font that supports those symbols, an encoding that reaches those glyphs, and a little bit of CONTEXT code to organise those symbols into symbol sets. As there are hundreds of these symbols, it's quite fortunate that this process is scriptable.

OpenType fonts formalise support for UNICODE, whether they be in TrueType (`ttf`) or PostScript (`otf`) glyph format. As such, this article can be seen as an extension of OpenType support in the script dimension.[1]

## Getting the right encoding

TEX, rather infamously, is still saddled with an 8-bit limit when dealing with fonts. So a TEX font can only contain 256 glyphs. Supporting UNICODE fonts thus means subdividing a large font with over a thousand symbols into 256-glyph chunks. The particular glyphs in a chunk are identified by their postscript names, and this collection of 256 glyph names constitute an encoding, designated with an `.enc` file suffix.

Getting an encoding right is a bit of an art. The mapping from UNICODE name to postscript name is different with each font: there is no standard postscript name for most UNICODE entities. The approach for getting an encoding depends on how the font encodes its constituent glyphs. A simple indicator of what can be found within a font is by looking at the `afm` file. The glyph names are typically mostly named or mostly numbered.

### Sequential Encoding

The first way that fonts can identify their glyph names is sequentially, by index. It's especially helpful if a font identifies glyphs in UNICODE order. Many of Adobe's OpenType fonts do this, with glyphs accessible from names like uni0041 and uni222A, the hexadecimal numbers referring directly to the unicode glyphs at the corresponding number.

An encoding can be synthesised directly with a tiny PERL script, `unienc.pl`:

```
#!/usr/bin/perl

print "/Unicode_$ARGV[0]_Encoding[ \n";

for ($n=0;$n<256;$n++)
  { printf ("/uni$ARGV[0]\U%02x\n",$n) }
```

```
print "] def\n";
```

There are other fonts that enumerate the glyphs in font order, rather than UNICODE or any other order, and, worse, don't give any meaningful information in the glyph names. The `Apple Symbols.ttf` font was like this, with glyphs labelled as gid65 and gid1146. A simple modification of the above PERL script will gladly spit out 256 sequential gid–prefixed glyph names. This is useful for accessing non-UNICODE glyphs in a font.

### Named Encoding

If a font's glyphs are mostly named, then one can laboriously assemble an encoding by hand. It would be more useful if that process can be scripted. In order to do so, some mapping from UNICODE number to glyph name must be obtained. Non-standard font manipulation tools must be used for that.

Apple provides one such tool in their FTXTOOLS suite.[2] It can dump and manipulate fonts using XML as a data format. An XML dump of a font's `cmap` table is just what's needed for inspecting the character mapping. The command is:

```
ftxdumperfuser -A d -t cmap -u -p -n fontfile.ttf
```

Of more general use are the TTX FontTools, from Just van Rossum/LettError.[3] It also dumps and manipulates fonts using XML as an interchange format. In order to get a minimally useful `.ttx` file, a command would be:

```
ttx -t cmap -t name fontfile.ttf
```

This yields an XML file like:

```
<ttFont sfntVersion="\x00\x01\x00\x00" ttLibVersion="2.0b1">
    <cmap>
      <tableVersion version="0"/>
      <cmap_format_4 platformID="3" platEncID="1" version="0">
        <map code="0x2600" name="gid289"/>
        <map code="0x2601" name="gid290"/>
        <map code="0x2602" name="gid291"/>
        <map code="0x2603" name="gid292"/>
        <map code="0x2604" name="gid293"/>
      </cmap_format_4>
    </cmap>
    <name>
      <namerecord nameID="1" platformID="1" platEncID="0" langID="0x0">
        Apple Symbols
      </namerecord>
    </name>
  </ttFont>
```

It's pretty clear from inspection how the file relates UNICODE numbers (`codes`) with POSTSCRIPT glyph names (`names`). Not every font makes all of the necessary tables visible, so other strategies need to be used in those cases. If a complete `.ttx` file is available, however, then you can use `ttx2enc.xsl`, a stylesheet that transforms a TTX file into an `enc` file for use with TEXFONT and PDFTEX:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version=
"1.0" xmlns:date="http://exslt.org/dates-and-times">
    <xsl:output method="text"/>
    <xsl:strip-space elements="*"/>
    <xsl:param name="vector">
    </xsl:param>
    <xsl:variable name="hexdigits" select="'0123456789abcdef'"/>
    <xsl:template name="grab-glyph-name">
      <xsl:param name="char-value"/>
      <xsl:choose>
```

```
    <xsl:when test="map[@code = $char-value]">
      <xsl:text>/  </xsl:text>
      <xsl:value-of select="map[@code = $char-value]/@name"/>
      <xsl:text> %   </xsl:text>
      <xsl:value-of select="$char-value"/>
      <xsl:text>   </xsl:text>
      <xsl:value-of select="map[@code = $char-value]/following-sibling::comment()[1]"/>
      <xsl:text>   </xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>/.notdef %   </xsl:text>
      <xsl:value-of select="$char-value"/>
      <xsl:text>   </xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<xsl:template match="/">
  <xsl:text>% Automatically generated encoding from ttx2enc.xsl % ttx2enc.xsl
by Adam T. Lindsay, 2004-01-23 % generated on   </xsl:text>
  <xsl:value-of select="date:date-time()"/>
  <xsl:text> % for the font:    </xsl:text>
  <xsl:value-of select="normalize-space(/ttFont/name/namerecord[@nameID='1'][1])"/>
  <xsl:text>   </xsl:text>
  <xsl:value-of select="normalize-space(/ttFont/name/namerecord[@nameID='2'][1])"/>
  <xsl:text> % for the vector:    </xsl:text>
  <xsl:value-of select="concat('0',$vector)"/>
  <xsl:text> /Unicode  </xsl:text>
  <xsl:value-of select="concat('0',$vector)"/>
  <xsl:text>Encoding [   </xsl:text>
  <xsl:apply-templates mode="cmap2glyph"/>
  <xsl:text>] def   </xsl:text>
</xsl:template>
<xsl:template match="/ttFont/cmap/cmap_format_4[1]" mode = "cmap2glyph">
  <xsl:call-template name="iterator"/>
</xsl:template>
<xsl:template match="*"/>
<xsl:template match="text()" mode="cmap2glyph"/>
<xsl:template name="iterator">
  <xsl:param name="outervalue">
    0
  </xsl:param>
  <xsl:param name="innervalue">
    0
  </xsl:param>
  <xsl:call-template name="grab-glyph-name">
    <xsl:with-param name="char-value">
    <xsl:value-of select = "concat('0x', $vector, $outervalue, $innervalue)"/>
    </xsl:with-param>
  </xsl:call-template>
  <xsl:choose>
    <xsl:when test="$outervalue='f' and $innervalue='f'">
    </xsl:when>
    <xsl:when test="$innervalue='f' and $outervalue!='f'">
      <xsl:call-template name="iterator">
        <xsl:with-param name="outervalue">
        <xsl:value-of select = "substring($hexdigits, string-length(
      substring-before( $hexdigits, $outervalue))+2, 1)"/>
        </xsl:with-param>   <xsl:with-param name="innervalue">0  </xsl:with-param>
      </xsl:call-template>
    </xsl:when>
```

```
        <xsl:otherwise>
          <xsl:call-template name="iterator">
            <xsl:with-param name="outervalue">
            <xsl:value-of select = "$outervalue"/>
            </xsl:with-param>    <xsl:with-param name="innervalue">
            <xsl:value-of select = "substring( $hexdigits, string-length(
          substring-before( $hexdigits, $innervalue))+2, 1)"/>
            </xsl:with-param>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:template>
  </xsl:stylesheet>
```

(For MacOSX users, the included `cmap2enc.xsl` transform will accomplish the same thing, but with a `.cmap.xml` file input. This one was written first, in fact.)

The UNICODE `enc` files for one font were generated with different values for the `vector` parameter. The vector identifies the high byte of the UNICODE value, and therefore the group of 256 glyphs. Which vectors are interesting can be discovered through inspection of the font and by looking at the UNICODE file `Blocks.txt`.[4] For example, the "Miscellaneous Symbols" block is located in the range 2600 to 26FF, corresponding with vector 26.

Michael Kay's SAXON[5] was the XSLT processor of choice, but you can use the command of your choice:

```
saxon AppleSymbols.ttx ttx2enc.xsl vector=1e > applesymbols1exx.enc
```

Note, above, that the both the `vector` parameter and the name of the output `enc` file should be all lowercase. This yields a file like:

```
/Unicode26Encoding [
/gid289 % 0x2600 BLACK SUN WITH RAYS
/gid290 % 0x2601 CLOUD
/gid291 % 0x2602 UMBRELLA
/gid292 % 0x2603 SNOWMAN
/gid293 % 0x2604 COMET
        % ... 250 more glyphs ...
/.notdef % 0x26FF
] def
```

### Perl from Unicode

The methods described above are only a couple possibilities. Another one is to use the `UnicodeData.txt`[6] file with PERL, RUBY, or another text processor of your choice.

## Font installation

Be sure to install any newly generated encodings into a place where `kpsewhich` can find them, like `texmf-fonts/dvips/local/`.

Once you have the encoding files for the vectors of interest in place, you can run TEXFONT with the appropriate encoding, like:

```
texfont --make --install --ve=foo --co=bar --en=bar26xx
```

## Support file

CONTEXT has very nice support for named symbols which can be loaded by named sets. It makes sense to use this mechanism. As these symbols are UNICODE entities, we may as well call UNICODE `\uchar` commands directly. As we have hundreds and

hundreds of glyphs to deal with, it makes sense to script the assignment of names, as well.

A modification of the Apple XSLT stylesheet (included in the distribution as `cmap2symb.xsl`) will take a vector and convert the all-caps unicode names in a `.cmap.xml` file into inter-capped symbol names within a symbol definition:

This results in ouput lines like the following:

```
\definesymbol[BlackSunWithRays][\uchar{38}{0}] % BLACK SUN WITH RAYS
\definesymbol[Cloud][\uchar{38}{1}] % CLOUD
\definesymbol[Umbrella][\uchar{38}{2}] % UMBRELLA
\definesymbol[Snowman][\uchar{38}{3}] % SNOWMAN
```

A little bit of manual effort to group the symbols into symbol sets followed, along with some shortening and correction of names. In general, the sets are named after the corresponding UNICODE block- or sub-block. The sets defined so far are as follows:

| **Symbol Set Name** | **Block** | **Required Font** |
|---|---|---|
| Unicode Additional Punctuation | 0x2000 | UnicodeRegular20 |
| Unicode Currency | 0x20A0 | UnicodeRegular20 |
| Unicode Letterlike | 0x2100 | UnicodeRegular21 |
| Unicode Letterlike Additional | 0x2100 | UnicodeRegular21 |
| Unicode Script Letterlike | 0x2100 | UnicodeRegular21 |
| Unicode Hebrew Letterlike | 0x2100 | UnicodeRegular21 |
| Unicode Turned Letterlike | 0x2100 | UnicodeRegular21 |
| Unicode Black-letter Letterlike | 0x2100 | UnicodeRegular21 |
| Unicode Double-struck Letterlike Math | 0x2100 | UnicodeRegular21 |
| Unicode Roman Numerals | 0x2150 | UnicodeRegular21 |
| Unicode Small Roman Numerals | 0x2150 | UnicodeRegular21 |
| Unicode Arrows | 0x2190 | UnicodeRegular21 |
| Unicode Multi Arrows | 0x2190 | UnicodeRegular21 |
| Unicode Optical Character Recognition | 0x2440 | UnicodeRegular24 |
| Unicode Circled Digits | 0x2460 | UnicodeRegular24 |
| Unicode Box Drawing | 0x2500 | UnicodeRegular25 |
| Unicode Double Box Drawing | 0x2500 | UnicodeRegular25 |
| Unicode Block Elements | 0x2580 | UnicodeRegular25 |
| Unicode Shade Characters | 0x2580 | UnicodeRegular25 |
| Unicode Terminal Graphics | 0x2580 | UnicodeRegular25 |
| Unicode Geometric Shapes | 0x25A0 | UnicodeRegular25 |
| Unicode Control Code Graphics | 0x25A0 | UnicodeRegular25 |
| Unicode Weather and Astrological | 0x2600 | UnicodeRegular26 |
| Unicode Miscellaneous | 0x2600 | UnicodeRegular26 |
| Unicode Japanese Chess | 0x2600 | UnicodeRegular26 |
| Unicode Pointing Hand | 0x2600 | UnicodeRegular26 |
| Unicode Warning Signs | 0x2600 | UnicodeRegular26 |
| Unicode Healing Signs | 0x2600 | UnicodeRegular26 |
| Unicode Religious and Political | 0x2600 | UnicodeRegular26 & 27 |
| Unicode Trigram | 0x2600 | UnicodeRegular26 |
| Unicode Zodiac | 0x2600 | UnicodeRegular26 |
| Unicode Chess | 0x2600 | UnicodeRegular26 |
| Unicode Playing Card | 0x2600 | UnicodeRegular26 |
| Unicode Musical | 0x2600 | UnicodeRegular26 |
| Unicode Recycling | 0x2600 | UnicodeRegular26 |
| Unicode Dice | 0x2600 | UnicodeRegular26 |
| Unicode Go Markers | 0x2600 | UnicodeRegular26 |
| Unicode Dingbats | 0x2700 | UnicodeRegular27 |

| | | |
|---|---|---|
| Unicode Checks and Xs | 0x2700 | UnicodeRegular27 |
| Unicode Stars | 0x2700 | UnicodeRegular27 |
| Unicode Snowflakes | 0x2700 | UnicodeRegular27 |
| Unicode Shadowed Shapes | 0x2700 | UnicodeRegular27 |
| Unicode Bars | 0x2700 | UnicodeRegular27 |
| Unicode Dingbat Punctuation | 0x2700 | UnicodeRegular27 |
| Unicode Hearts | 0x2700 | UnicodeRegular27 |
| Unicode Negative Circled Digits | 0x2700 | UnicodeRegular27 |
| Unicode Circled Sans-serif Digits | 0x2700 | UnicodeRegular27 |
| Unicode Negative Circled Sans-serif Digits | 0x2700 | UnicodeRegular27 |
| Unicode Dingbat Arrows | 0x2700 | UnicodeRegular27 |
| Unicode Shadowed Arrows | 0x2700 | UnicodeRegular27 |
| Unicode Tailed Arrows | 0x2700 | UnicodeRegular27 |

## Usage

In order use these pre-defined symbols, load the definitions from the somewhat presumptuously named `symb-uni.tex`:

```
\usesymbols[uni]
```

As the symbol definitions depend on unicode fonts being defined, you need to load map files and define simple font synonyms:

```
\loadmapfile [applesymbols25xx-apple-applesymbols.map]
\loadmapfile [applesymbols26xx-apple-applesymbols.map]
\definefontsynonym [UnicodeRegular25] [applesymbols25xx-AppleSymbols]
\definefontsynonym [UnicodeRegular26] [applesymbols26xx-AppleSymbols]
```

Using the symbols is like others in ConTeXt.

```
\showsymbolset[Unicode Warning Signs][n=4]
```

SkullAndCrossbones CautionSign RadioactiveSign BiohazardSign

```
\setupsymbolset[Unicode Healing Signs]
Here's an Ankh: \symbol [Ankh] \quad and a Caduceus: \symbol [Caduceus]
```

Here's an Ankh: ☥ and a Caduceus: ⚚

```
Here's a hammer and sickle, called without loading the symbol set:
\symbol [Unicode Religious and Political] [HammerAndSickle]
```

Here's a hammer and sickle, called without loading the symbol set: ☭

```
\definesymbol[1][{\symbol[Unicode Miscellaneous][BallotBoxWithCheck]}]
\definesymbol[2][{\symbol[Unicode Pointing Hand][WhiteRightPointingIndex]}]
\startitemize[packed]
\item You can hook symbols into the itemize mechanism at different levels.
\startitemize[packed]
  \item This is not new, but you have many more typographic options
  available to you.
  \item Be sure to use typographic restraint and good taste!
\stopitemize
\stopitemize
```

☑ You can hook symbols into the itemize mechanism at different levels.

☞ This is not new, but you have many more typographic options available to you.
☞ Be sure to use typographic restraint and good taste!

## Notes

1. see other articles at http://homepage.mac.com/atl/tex/.
2. available at http://developer.apple.com/fonts/OSXTools.html
3. available at http://sourceforge.net/projects/fonttools/
4. from http://www.unicode.org/Public/UNIDATA/
5. from http://saxon.sourceforge.net/
6. from http://www.unicode.org/Public/UNIDATA/

Adam T. Lindsay
Lancaster University