Unicode introduction

Ken Zook
October 20, 2008

Contents

1 Unicode introduction 1
2 Unicode properties (slide 1) 1
3 Unicode code space (slide 2) 2
4 Encoding Unicode (slide 3) 2
5 Private Use Area (slide 4) 3
6 Canonical equivalence (slide 6) 4
7 Normalization (NFD) (slide 7) 5
8 Normalization (NFC) (slide 8) 6
9 Case conversion (slides 9–12) 7
10 Smart rendering: Arabic (slide 13) 9
11 Smart rendering: Burmese (slide 14) 9
12 Smart rendering: Tamil (slide 15) 9
13 FieldWorks rendering 9

1 Unicode introduction

After the punch card and paper tape days, computers standardized on the ASCII character set which encoded 127 control codes and basic English characters. With the advent of Windows, the standard became the ANSI character set which encoded 255 control codes and characters to cover most common European languages. As the use of computers became more global, there was a critical need to standardize a system that could encode all languages of the world. Unicode was developed to meet this need and is now the main standard for encoding multilingual data on computers. The primary Web site for Unicode is www.unicode.org. The Unicode Consortium meets regularly to review new requests for code points in the standard, and every couple years or so a new version of Unicode is approved with additional code points. The current version is 5.1 which defines around 99,000 code points covering most major languages of the world.

2 Unicode properties (slide 1)

The Unicode standard not only defines the character (or representative glyph) for a code point, but also includes semantic properties for the character. These properties include information to indicate whether the character is a letter, a digit, or punctuation. It also includes information such as case mapping, normalization, directional, and collation information.

The Unicode Consortium maintains master ASCII tables with this information that are available online at www.unicode.org/onlinedat/online.html. The primary table is UnicodeData.txt. After installing FieldWorks (or SIL Encoding Converters), users can view the SIL version of this table at %ALLUSERSPROFILE%\Application Data\SIL\Icu36\data\unidata. This file has one line per code point with 15 semicolon-

delimited fields. See http://www.unicode.org/Public/UNIDATA/UCD.html for a description of these properties.

Note: %ALLUSERSPROFILE%\Application Data is c:\Documents and Settings\All Users\Application Data on Windows XP and c:\ProgramData on Vista.

Example letter "A"

```
0041;LATIN CAPITAL LETTER A;Lu;0;L;;;;;N;;;;0061;
```

Code point: 0041
Name: LATIN CAPITAL LETTER A
General category: Uppercase letter (Lu)
Canonical combining class: Standard spacing (0)
Bidi category: Left-to-right (L)
Mirrored: no (N)
Lowercase mapping: 0061

3 Unicode code space (slide 2)

Unicode provides code points for over 1 million characters (1,111,998 assignable code points). The code space ranges from 0000 to 10FFFF, arranged in 17 planes of 64K code points each.

Examples

- Plane 0 (also known as the Basic Multilingual Plane or BMP) includes 0000–FFFF.
- Plane 1 includes 10000–1FFFF.

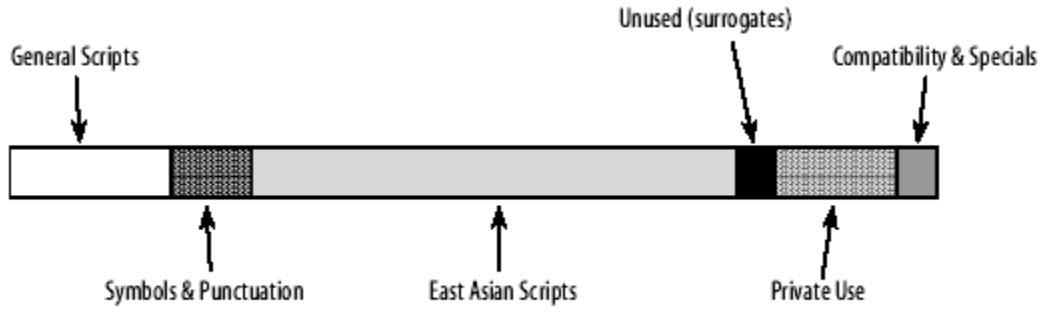

Figure 1. Organisation of the BMP

4 Encoding Unicode (slide 3)

Unicode can be encoded as

- UTF-32 (one 32-bit value per code point)
- UTF-16 (one or two 16-bit values per code point), or
- UTF-8 (one to four 8-bit values per code point).

UTF-16 also comes in Big Endian (BE) and Little Endian (LE) variants depending on your operating system. XML files typically use UTF-8, and FieldWorks and most Windows applications use UTF-16(LE). Go to http://rishida.net/scripts/uniview/conversion to convert between different modes.

In order to reach code points in Planes 1–16 in UTF-16, Unicode reserves 2,048 code points, called surrogates, in the range D800–DFFF. Surrogates always come in pairs: a high surrogate (D800–DBFF) followed by a low surrogate (DC00–DFFF). Since Unicode 3.1, assignments have been made outside of the BMP that require surrogates.

Example

```
10331 GOTHIC LETTER BAIRKAN in UTF-16 = D800 DF31
```

FieldWorks and the database use UTF-16. To encode code points in Plane 0 (0000–FFFF) it uses a single 16-bit value, but for all other code points it uses two 16-bit surrogate values.

If N is the UTF-32 value, H is the high surrogate, and L the low surrogate, users can convert surrogates to UTF-32 using the following formula:
N = (H - D800) * 400 + (L - DC00) + 10000

To convert to UTF-16 surrogates from UTF-32, use the following formulas:
H = (N - 10000) / 400 + d800 (or >> 10 instead of / 400)
L = (N - 10000) mod 400 + dc00 (or & 3ff instead of mod 400)

Binary representation of UTF-8 characters

32 bit Hex Range	UTF-8 Binary representations
00000000-0000007f	0xxxxxxx (ASCII)
00000080-000007ff	110xxxxx 10xxxxxx
00000800-0000ffff	1110xxxx 10xxxxxx 10xxxxxx
00010000-001fffff	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
00200000-03ffffff	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
04000000-7fffffff	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Note: In UTF-8, any character that has the upper bit set must be a part of a multibyte character. As a result, an upper ANSI character in a UTF-8 file will be flagged as an illegal character. This format was designed so that regardless of where a pointer is set in a UTF-8 string, the software can easily find the beginning byte of the sequence.

5 Private Use Area (slide 4)

The Unicode Consortium defines most of the characters in Unicode. There are some code points, however, which they will never assign. These are reserved for users or vendors to use in any way they want, and are called the Private Use Areas (PUA).

PUA Areas

```
E000–F8FF (6,400 code points in the BMP)
F0000–FFFFD (65,534 code points in Plane 15)
100000–10FFFD (65,534 code points in Plane 16)
```

For use within SIL, and our Family of Organizations, NRSI has subdivided the BMP PUA code points into two sections:

```
E000 – EFFF (4096 code points) Entities can define as needed
F100 – F8FF (2048 code points) Assigned by NRSI for international use
```

(The code points between these two sections have been used by Microsoft products, so SIL is avoiding them.) The hope is that entities will work with NRSI to get international assignments for all code points they need. When this is not possible, they can use the entity section. But when they do this, NRSI will also assign PUA code points in Plane 15

or 16 that uniquely identify every code point assigned by individual entities. This provides a resolution to the problem where two entities define the same code point in different ways. If the data must be shared outside the entity, one or more of the data sets will need to translate their entity-assigned code points to the corresponding unique Plane 15-16 code points.

Example

```
E010 assigned in the Philippines (may uniquely map to F3010)
E010 assigned in Russia (may uniquely map to F1010)
```

In order to fully handle Unicode, SIL must handle surrogates and PUA seamlessly. FieldWorks uses ICU (see ICU and writing systems.doc) to provide this capability.

6 Canonical equivalence (slide 6)

In addition to the problems with surrogates and PUA, another challenge is the way Unicode defined code points for some characters. In order to maintain compatibility with existing code pages, Unicode assigned numerous composite characters (base character + one or more combining marks). However, the same characters can also be encoded as a base character plus one or more combining marks. Furthermore, in some situations, there can be several valid orders for combining marks. These ways of representing the same typographic character are considered canonically equivalent, and software must treat them as equivalent.

Example

A Latin capital letter A with ring above and acute can be represented by the following code points:

```
01FA
212B 0301
00C5 0301
0041 030A 0301
```

```
0041;LATIN CAPITAL LETTER A;Lu;0;L;;;;;N;;;;0061;
00C5;LATIN CAPITAL LETTER A WITH RING ABOVE;Lu;0;L;0041 030A;;;;N;
    LATIN CAPITAL LETTER A RING;;;00E5;
01FA;LATIN CAPITAL LETTER A WITH RING ABOVE AND ACUTE;Lu;0;L;
    00C5 0301;;;;N;;;;01FB;
0301;COMBINING ACUTE ACCENT;Mn;230;NSM;;;;;N;NON-SPACING ACUTE;Oxia, Tonos;;;
030A;COMBINING RING ABOVE;Mn;230;NSM;;;;;N;NON-SPACING RING ABOVE;;;;
212B;ANGSTROM SIGN;Lu;0;L;00C5;;;;N;ANGSTROM UNIT;;;00E5;
```

From a software standpoint, things such as display, cursor movement, searching, and sorting must work identically regardless of which form is used. For software to sort or search data, the simplest solution is to first normalize the data so it is identical regardless of the original state, thus allowing for efficient algorithms. There are several different ways to normalize but these are the two most common:

- Canonical decomposition (NFD)
- Canonical composition (NFC)

Go to http://www.unicode.org/reports/tr15/ for a detailed description of Unicode normalization.

A program that fully supports Unicode is supposed to treat canonical equivalent combinations seamlessly so that it doesn't matter to the user which version is being used. Searching and sorting should work identically regardless of the underlying code points.

FieldWorks attempts to fully implement Unicode so that it doesn't matter to the user. To do this, it maintains data consistently in NFD and anything that is exported or placed on the clipboard is switched to NFC. The main place this breaks down is when outside software interfaces directly with internal Fieldworks data. Examples are Keyman and Bulk edit processors. In these cases the designer of the Keyman table or the Bulk edit processor (e.g., CC table) must be prepared to handle NFD data. Most other Unicode programs are not as rigorous as FieldWorks. Microsoft Word, for example, does not normalize data, and if you search for 1F71 it will not find 03AC or 03B1 0301 like it should if it were properly implementing the Unicode standard.

Although software that fully supports Unicode normalization treats any of the canonical forms seamlessly, few programs reach this standard.

7 Normalization (NFD) (slide 7)

Canonical decomposition (NFD) is the process of

- taking a string
- recursively replacing composite characters using the Unicode canonical decomposition mappings (including algorithmic Hangul canonical decomposition mappings), and
- putting the result in canonical order.

A string is put into canonical order by repeatedly replacing any exchangeable pair by the pair in reversed order. When there are no remaining exchangeable pairs, the string is in canonical order.

A sequence of two adjacent characters in a string is an exchangeable pair if the combining class (from the Unicode Character Database) for the first character is greater than the combining class for the second, and the second is not a starter.

Here is a section of UnicodeData.txt. Notice the Character Decomposition Mapping (field 5) and Canonical Combining Class property (field 3):

```
006F;LATIN SMALL LETTER O;Ll;0;L;;;;;N;;;004F;;004F
014D;LATIN SMALL LETTER O WITH MACRON;Ll;0;L;006F 0304;;;;N;
   LATIN SMALL LETTER O MACRON;;014C;;014C
01EB;LATIN SMALL LETTER O WITH OGONEK;Ll;0;L;006F 0328;;;;N;
   LATIN SMALL LETTER O OGONEK;;01EA;;01EA
01ED;LATIN SMALL LETTER O WITH OGONEK AND MACRON;Ll;0;L;01EB 0304;;;;N;
   LATIN SMALL LETTER O OGONEK MACRON;;01EC;;01EC
0304;COMBINING MACRON;Mn;230;NSM;;;;;N;NON-SPACING MACRON;;;;
0328;COMBINING OGONEK;Mn;202;NSM;;;;;N;NON-SPACING OGONEK;;;;
```

Decomposition example

006F 0328 0304
006F 0304 0328 ≡ 006F 0328 0304
01EB 0304 ≡ 006F 0328 0304
014D 0328 ≡ 006F 0304 0328 ≡ 006F 0328 0304
01ED ≡ 01EB 0304 ≡ 006F 0328 0304

For the fourth example, see that 014D decomposes to 006F 0304. But then the combining class (230/202) is used to reorder the final two code points to produce the final order.

Note: In each instance, users end up with the same sequence of code points once in the NFD state.

FieldWorks stores all data in the database in NFD. In most cases, users do not need to worry about normalization because the program handles everything seamlessly. However, if writing converters for bulk edit, be prepared to handle NFD since this is the input to the processor. The output will be normalized regardless of the output of the processor. Also, when designing Keyman keyboards that involve surrounding context, the table must be prepared to deal with NFD data. With some additional work, a Keyman table can be designed to work with NFD data for FieldWorks as well as NFC data typically used in other applications.

FieldWorks normalizes data to NFC any time it puts something on the clipboard since most programs prefer this format. As a result, you won't see the internal NFD format by copying from FieldWorks to something else. By using bulk edit in Flex, you can see the internal code points using the Any to Unicode transducer. For example, if you paste 1F71 into a field, and then use the Any to Unicode transducer, you'll see that it's actually 3b1 301 (NFD) inside FieldWorks. But if you cut this character from FieldWorks and paste into Word or ZEdit to check the value, you'll see 3ac (NFC).

Due to some canonical equivalents and the Unicode normalization values, a few code points are not stable and will not survive round trips between NFD and NFC. These code points are rare. It is best not to use these code points in data. Here are two examples.

1F71 in NFD is 03B1 0301 or in NFC it is 03AC.

1FFD in NFD is 00B4 or in NFC it is also 00B4.

Normalization tables in Unicode were designed for modern day Hebrew. Using standard normalization on Biblical Hebrew results in incorrect reordering of diacritics that will damage data. To solve this problem, FieldWorks uses custom normalization of Hebrew as described in SBLHebrewUserManual1.5x.pdf available at http://www.sbl-site.org/educational/BiblicalFonts_SBLHebrew.aspx. The Society of Biblical Literature and SIL International agreed on a non-standard ordering for Biblical Hebrew that allows correct printing with Ezra SIL and SBL fonts. FieldWorks can process these texts without damaging the data through normalization. However, other programs that use standard Unicode normalization will likely damage this data.

8 Normalization (NFC) (slide 8)

Canonical composition is accomplished by decomposing the original strings, then using the Decomposition Mapping (field 5) in reverse order to compose the string as much as possible.

Example

006F 0328 0304 (NFD) ≡ 01EB 0304 ≡ 01ED
006F 0304 0328 ≡ 006F 0328 0304 (NFD) ≡ 01EB 0304 ≡ 01ED
01EB 0304 ≡ 006F 0328 0304 (NFD) ≡ 01EB 0304 ≡ 01ED
014D 0328 ≡ 006F 0304 0328 ≡ 006F 0328 0304 (NFD) ≡ 01EB 0304 ≡ 01ED
01ED ≡ 01EB 0304 ≡ 006F 0328 0304 (NFD) ≡ 01EB 0304 ≡ 01ED

Again, the result on the right is the same regardless of the starting codes.

In FieldWorks, when users dump data to XML or SFM, or copy text to the clipboard, it converts the data to NFC, since many programs assume this normalization.

9 Case conversion (slides 9–12)

Because of compatibility issues, Unicode contains some code points that are digraphs, which hold more than one typographical character. Because of these, there are actually three case states: lowercase, uppercase, and titlecase. Detection of case, conversion to upper/lower/title case, and caseless matching are all much more complex than in simple ASCII days.

Go to http://www.unicode.org/reports/tr21/ for a detailed description of Unicode case mappings.

The full case mappings for Unicode characters are obtained by using the SpecialCasing.txt mappings plus the UnicodeData.txt mappings, excluding any latter mappings that would conflict. Any character that does not have a mapping in these files is considered to map to itself.

Example

```
01F1;LATIN CAPITAL LETTER DZ;Lu;0;L;<compat> 0044 005A;;;;N;;;;01F3;01F2
01F2;LATIN CAPITAL LETTER D WITH SMALL LETTER Z;Lt;0;L;
    <compat> 0044 007A;;;;N;;;01F1;01F3;
01F3;LATIN SMALL LETTER DZ;Ll;0;L;<compat> 0064 007A;;;;N;;;01F1;;01F2
```

UnicodeData.txt contains the following information related to case mapping:

- General Category (Field 2) (Lu = Letter, Uppercase, Ll = Letter, Lowercase, Lt = Letter, Titlecase)
- Uppercase Mapping (Field 12)
- Lowercase Mapping (Field 13)
- Titlecase Mapping (Field 14)

Case mapping is not reversible. For example, “McConnel” would uppercase to “MCCONNEL” but lowercase to “mcconnel” and titlecase to “Mcconnel”. None match the original string.

Case mappings may produce strings of different length from the original:

```
01F0;LATIN SMALL LETTER J WITH CARON;Ll;0;L;006A 030C;;;;N;
    LATIN SMALL LETTER J HACEK;;;;
004A;LATIN CAPITAL LETTER J;Lu;0;L;;;;;N;;;;006A;
030C;COMBINING CARON;Mn;230;NSM;;;;;N;NON-SPACING HACEK;;;;
```

For example, lowercase j with caron (ǰ)is a combined character (01F0). However, there is not a composite of uppercase J with caron, so the uppercase equivalent of 01F0 is 004A, 030C.

Casing conversions may be language (locale) dependent:

```
0069;LATIN SMALL LETTER I;Ll;0;L;;;;;N;;;0049;;0049
0049;LATIN CAPITAL LETTER I;Lu;0;L;;;;;N;;;;0069;
0130;LATIN CAPITAL LETTER I WITH DOT ABOVE;Lu;0;L;0049 0307;;;;N;
    LATIN CAPITAL LETTER I DOT;;;0069;
```

In English, the uppercase of ‘i’ is ‘I’, but in Turkish and Azeri, the uppercase equivalent is 0130, which has a dot above an uppercase ‘I’.

Case mapping may depend on context. SpecialCasing.txt is a Unicode data file that lists exceptions to casing rules that can’t be handled by the standard UnicodeData.txt file. For example, 03A3 "Σ" capital sigma lowercases to 03C3 "σ" small sigma if it is followed by another letter, but lowercases to 03C2 "ς" small final sigma if it is not.

Some characters require special handling, such as 0345:

```
0345;COMBINING GREEK YPOGEGRAMMENI;Mn;240;NSM;;;;;N;
    GREEK NON-SPACING IOTA BELOW;;0399;;0399
1F00;GREEK SMALL LETTER ALPHA WITH PSILI;Ll;0;L;03B1 0313;;;;N;;;1F08;;1F08
1F81;GREEK SMALL LETTER ALPHA WITH DASIA AND YPOGEGRAMMENI;Ll;0;L;
    1F01 0345;;;;N;;;1F89;;1F89
03B1;GREEK SMALL LETTER ALPHA;Ll;0;L;;;;;N;;;0391;;0391
0314;COMBINING REVERSED COMMA ABOVE;Mn;230;NSM;;;;;N;
    NON-SPACING REVERSED COMMA ABOVE;Dasia;;;
1F08;GREEK CAPITAL LETTER ALPHA WITH PSILI;Lu;0;L;0391 0313;;;;N;;;;1F00;
0391;GREEK CAPITAL LETTER ALPHA;Lu;0;L;;;;;N;;;;03B1;
0313;COMBINING COMMA ABOVE;Mn;230;NSM;;;;;N;NON-SPACING COMMA ABOVE;Psili;;;
0399;GREEK CAPITAL LETTER IOTA;Lu;0;L;;;;;N;;;;03B9;
1F88;GREEK CAPITAL LETTER ALPHA WITH PSILI AND PROSGEGRAMMENI;Lt;0;L;
    1F08 0345;;;;N;;;;1F80;
1F80;GREEK SMALL LETTER ALPHA WITH PSILI AND YPOGEGRAMMENI;Ll;0;L;
    1F00 0345;;;;N;;;1F88;;1F88
03B1;GREEK SMALL LETTER ALPHA;Ll;0;L;;;;;N;;;0391;;0391
0391;GREEK CAPITAL LETTER ALPHA;Lu;0;L;;;;;N;;;;03B1;
03B9;GREEK SMALL LETTER IOTA;Ll;0;L;;;;;N;;;0399;;0399
1FBF;GREEK PSILI;Sk;0;ON;<compat> 0020 0313;;;;N;;;;;
```

Lowercase Greek alpha with an acute accent and an iota subscript (1F80) maps to 1F88 where the iota becomes a lowercase base character when uppercased without following letters. When in the middle of a word, the following iota becomes capitalized. Also, if the lowercase alpha has the acute and iota subscript as combining marks, it needs to be fully decomposed. Thus, the combining iota gets pushed to the end since it will become a base character when uppercased.

Casing operations may not preserve normalization:

```
030C;COMBINING CARON;Mn;230;NSM;;;;;N;NON-SPACING HACEK;;;;
0323;COMBINING DOT BELOW;Mn;220;NSM;;;;;N;NON-SPACING DOT BELOW;;;;
01F0;LATIN SMALL LETTER J WITH CARON;Ll;0;L;006A 030C;;;;N;
    LATIN SMALL LETTER J HACEK;;;;
004A;LATIN CAPITAL LETTER J;Lu;0;L;;;;;N;;;;006A;
```

Original (NFC)	j◌̌◌̣	U+01F0 LATIN SMALL LETTER J WITH CARON, U+0323 COMBINING DOT BELOW
Uppercased	J◌̌◌̣	U+004A LATIN CAPITAL LETTER J, U+030C COMBINING CARON, U+0323 COMBINING DOT BELOW
Uppercased NFC	J◌̣◌̌	U+004A LATIN CAPITAL LETTER J, U+0323 COMBINING DOT BELOW, U+030C COMBINING CARON,

The original string is in NFC format. When uppercased, the small j with caron turns into an uppercase J with a separate combining caron. If followed by a *below* combining mark,

it loses its normalization. The combining marks have to be put in canonical order for it to be normalized.

10 Smart rendering: Arabic (slide 13)

The Unicode standard provides one code point for each letter, regardless of the shape of the glyph. In some languages such as Arabic, the shape changes depending on the context. Unicode assumes there will be a smart rendering engine that will make these contextual changes as the code points are displayed.

In Arabic, type a *beh* and it inserts code point 0628 and displays the independent glyph. Enter a *fatha* vowel and it displays centered above the beh. Now type a second beh and the rendering engine changes the first beh to the initial glyph and moves the fatha appropriately. It then uses the final glyph for the second beh. Type a *kasra* vowel and it is positioned under the second beh. Type a third beh and the second beh switches to the medial glyph and the kasra is moved appropriately, then the third beh takes on the final form. Type a space and a fourth beh and it takes on the independent glyph.

Note: The code points are in logical order, allowing the rendering engine to position the characters appropriately from right to left.

In the early days this would require fancy keyboarding schemes to pick an appropriate code point for each contextual shape, and possibly reverse the code points in order to get right-to-left ordering. The former approach also had complexities when doing searching, sorting, or spell checking because the same underlying character was represented by four different code points, and the order might even be reversed.

11 Smart rendering: Burmese (slide 14)

Contextual rendering can also be illustrated by Burmese, where the following characters are rendered around the first consonant. First enter a *ka* consonant. Then enter the *ra* consonant, which needs to wrap around the ka. This is accomplished by inserting a *virama* (1039) in front of the ra. Then type the *u* vowel and it slips into the lower right corner, adjusting the ra accordingly. Finally, type the *i* vowel and it slips into the top right corner, again adjusting the ra accordingly.

12 Smart rendering: Tamil (slide 15)

A final example is from Tamil. Here the *uu* vowel has an independent form (0b8a) and a diacritic form (0bc2). The glyphs for many consonants as well as the uu change depending on the context. The first two examples use the ra following the independent form, and then preceding the diacritic form. Users can see how the glyphs change when the preceding consonant is a *ya, nna, ma, ka,* and *ja.*

Note: The code point for the diacritic uu is the same in every case, regardless of how the rendering engine displays the glyphs.

13 FieldWorks rendering

FieldWorks uses two smart rendering engines. The first comes from Microsoft and is their Uniscribe engine. This works with many standard Windows fonts such as Times

New Roman. Uniscribe supports most of the major languages around the world but does not support many minority languages that do not have a commercial value to make it profitable.

The second smart rendering engine in FieldWorks is the Graphite engine developed by NRSI to provide practically any smart rendering capabilities (except vertical rendering) needed by minority languages. See Rendering issues.doc for more detail.