UNICODE ATE MY BRAIN

John Cowan

Reuters Health Information

Copyright

- Copyright © 2001 John Cowan
- Licensed under the GNU General Public License
- ABSOLUTELY NO WARRANTIES; USE AT YOUR OWN RISK
- Portions written by Tim Bray; used by permission
- Title devised by Smarasderagd; used by permission
- Black and white for readability

Abstract

Unicode, the universal character set, is one of the foundation technologies of XML. However, it is not as widely understood as it should be, because of the unavoidable complexity of handling all of the world's writing systems, even in a fairly uniform way. This tutorial will provide the basics about using Unicode and XML to save lots of money and achieve world domination at the same time.

Roadmap

- Brief introduction (4 slides)
- Before Unicode (16 slides)
- The Unicode Standard (25 slides)
- Encodings (11 slides)
- XML (10 slides)
- The Programmer's View (27 slides)
- Points to Remember (1 slide)

How Many Different Characters?

a A à á â ã ä å ā ă ą

How Computers Do Text

- Characters in computer storage are represented by “small” numbers
- The numbers use a small number of bits: from 6 (BCD) to 21 (Unicode) to 32 (`wchar_t` on some Unix boxes)
- Design choices:
  - Which numbers encode which characters
  - How to pack the numbers into bytes

Where Does XML Come In?

- XML is a textual data format
- XML software is required to handle all commercially important characters in the world; a promise to “handle XML” *implies* a promise to be international
- Applications can do what they want; monolingual applications can *mostly* ignore internationalization

$$$ £££ ¥¥¥

- Extra cost of building-in internationalization to a new computer application: about 20% (assuming XML and Unicode).
- Extra cost of retrofitting internationalization into a monolingual application: about 100%.

BEFORE UNICODE

The Mess

- Each commercial culture developed its own way of representing characters
- The leaders in computing technology ignored the issue for much too long
- Data that had to cross cultural boundaries needed to be lowest-common-denominator or risk garbling

Character Sets

- Mappings between characters (for people) and code numbers (for computers)
- Also called “code pages”
- There are hundreds of them in use
- Neither ASCII nor Windows-1252 is universally used

ASCII

- A 7-bit character set, with 33 control characters, a space, and 94 printing characters
- An extension of the traditional U.S. typewriter keyboard
- Serves basic U.S. needs *only*

ISO 646-xx

- International version is just ASCII
- National versions replaced some ASCII characters with letters

Hello.c:

```
main(int argc, char *argv[]) {
  printf("Hello, world!\n");
}
```

ISO 646-xx

- International version is just ASCII
- National versions replaced some ASCII characters with letters

Hello.c in ISO-646-DK:

```
main(int argc, char *argvÆÅ) æ
  printf("Hello, world!Øn");
ã
```

ISO 8859-1 (Latin-1)

- An 8-bit upward compatible extension of ASCII
- Adds 96 additional characters
- Handles most Western European languages
- Windows-1252 adds 27 further characters

Alphabet Soup

- Latin-1 can’t do it all
- Central and Eastern European languages need Latin-2, which is only partly compatible
- Other languages need other parts of ISO 8859: Latin-3, Latin-4, Latin-5, … Latin-10.

ISO 8859: Mixed Alphabets

- These character sets are ASCII in the lower part, some other script in the higher part:
  - Greek, Russian, Hebrew, Arabic, Thai
- There is a Windows code page for each, typically not compatible

Global Diversity

- How international text (Greek, in this case) interacts with non-international applications
- Greek is a simple case: it can be handled by an ISO 8859 part
- Other languages make life far more difficult, as we shall see!

Excerpt from a Greek-language Home Page

Τυγχάνω ερευνητής στο κέντρο *Thesaurus Linguae Graecae (Θησαυρός Γλώσσης της Ελληνικής),* του Πανεπιστημείου της Καλιφορνίας στο Irvine --- και συνάμα (όπως φυσικά θα σας αποδείξουν και οι σελίδες μου) πρόσωπο ουχί ελλάσονος ενδιαφέροντος!

The Latin-1-only View

Ôõã÷Üíù åñåõíçôÞò óôï êÝíôñï *Thesaurus Linguae Graecae (Èçóáõñüò Ãëþóóçò ôçò ÅëëçíéêÞò)*, ôïõ Ðáíåðéóôçìåßïõ ôçò Êáëéöïñíßáò óôï Irvine --- êáé óõíÜìá (üðùò öõóéêÜ èá óáò áðïäåßîïõí êáé ïé óåëßäåò ìïõ) ðñüóùðï ïõ÷ß åëëÜóïíïò åíäéáöÝñïíôïò!

Frangovlakhika

Tugxanw ereunhths sto kentro *Thesaurus Linguae Graecae (Qhsauros Glwsshs ths Ellhnikhs),* tou Panepisthmeiou ths Kalifornias sto Irvine -- kai sunama (opws fusika qa sas apodeijoun kai oi selides mou) proswpo ouxi ellasonos endiaferontos!

(what he said)

I am a research associate at the *Thesaurus Linguae Graecae* in the University of California at Irvine, USA --- and, as this node will no doubt prove to you, an extremely interesting personage!

Problems of Specific Scripts

- Middle Eastern languages are written right-to-left, but must mix correctly with left-to-right text, either Latin or numbers
- South Asian languages have vowel marks that are sometimes written before (but always stored and pronounced after) the consonants

Problems of Specific Scripts

- East Asian writing systems use the huge (more than 50,000) set of Chinese characters or *hanzi*, often in combination with local scripts, large or small
- Go buy *CJKV Information Processing* by Ken Lunde (O'Reilly) if you care about the details

The Possibilities

- With ISO 8859, you can handle French *or* Hebrew *or* Greek,
- *or*, you can use JIS and handle Japanese, English, and Russian,
- *or*, you can use Big5 and handle Chinese and English…
- ISO 2022 allows mixing and matching at the cost of enormous complexity

THE UNICODE STANDARD

Mini-Roadmap

- Principles I
- Principles II
- Conformance
- Unicode Map

Principles I

- 21-bit character codes
- Efficiency
- Characters, not glyphs
- Well-defined semantics
- Plain text

Character Codes

- Unicode 4.0 has 57,129 16-bit characters out of a total maximum of 63,470
- A further 45,718 rare or archaic characters are encoded with two consecutive 16-bit code units from reserved ranges (called "surrogates")

Efficiency

- No special escape or shift characters required
- All representations of Unicode are self-synchronizing and can be randomly accessed
- Formatting characters are kept to a minimum

Characters vs. Glyphs

- Character: the smallest component of written language that has semantic value.
- Glyph: represents the shape of a character when rendered or displayed.
- Fonts contain glyphs, not characters

Characters vs. Glyphs

- Latin A and Greek A (alpha) are distinct characters with the same glyph
- Arabic letters need up to four glyphs (initial, medial, final, isolated)
- "f" plus "i" is rendered with a single merged glyph in fine typesetting

Well-defined Semantics

- Tables generated by the Unicode Consortium give the properties of characters
- Letter, number, punctuation mark, symbol, diacritic, whitespace …
- Case mapping, Arabic shaping, normalization ...

Plain Text

- Unicode encodes just enough information for *bare legibility*
- Plain text is public, standardized, and universally readable
- SGML, HTML, XML are suitable “fancy text” standards to supply structure and formatting to Unicode plain text

Principles II

- Logical ordering
- Unification
- Dynamic composition
- Equivalence
- Convertibility

Logical Ordering

- With one minor exception, characters are represented in Unicode in *logical order* (the order they are typed or spoken).
- Unicode provides a table-driven algorithm for reordering text into proper reading order, including mixed directions

Unification

- “A difference that *makes* no difference *is* no difference.” --Spock of Vulcan
- If characters look the same, and are from different source standards, they are a single Unicode character
- Common letters, punctuation marks, symbols, and diacritics are unified

Unification

- Differences in language, font, size, and positioning are not represented
- Identical-looking characters (a, alpha) from different scripts are *not* unified
- Characters that were distinct in a major national or industry standard are kept distinct for round-tripping purposes

Han Unification

- Chinese, Japanese, Korean all use the 3000-year-old Chinese characters (*hanzi, kanji, hanja*)
- Each national character set encodes the characters in its own way
- If it looks similar and is historically the same, Unicode unifies it!

Han Unification

- Unicode orders Han characters using the traditional Kang Xi dictionary and other dictionaries
- Language differences, which control the choice of fonts, are expressed by a higher-level protocol
- Simplified and traditional characters are *not* unified in Unicode

Dynamic Composition

- There is no character LATIN CAPITAL LETTER Q WITH CIRCUMFLEX
- It can be represented as LATIN CAPITAL LETTER Q followed by U+0302 COMBINING CIRCUMFLEX

Dynamic Composition

- COMBINING CIRCUMFLEX is not the same character as ASCII “^”
- Fonts can have a precomposed glyph for Q WITH CIRCUMFLEX

Equivalence

- Different ways of representing the same characters are equally valid
- Normalization forms allow documents to be compared easily by suppressing irrelevant encoding differences

Convertibility

- Characters in other character sets can be converted to and from Unicode, usually 1:1
- ASCII and Latin-1 map codepoint for codepoint
- Conversions done by mapping tables

Unicode General Categories

- Letters: upper, lower, title, modifier, other (syllables, ideographs, etc.)
- Numbers: digit, letter, other
- Punctuation: connector, dash, open, close, initial-quote, final-quote, other
- Marks: non-spacing, enclosing, other

Unicode General Categories

- Symbols: math, currency, modifier, other
- Separators: space, line, paragraph
- Other: control, format, surrogate, private-use

Unicode Map
Basic Multilingual Plane

- U+0xxx: ASCII, Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, Syriac, Thaana, Indic scripts, Thai, Lao, Tibetan
- U+1xxx: Myanmar, Georgian, Hangul, Ethiopic, Cherokee, Canadian Aboriginal, Ogham, Runic, Philippine scripts, Khmer, Mongolian, Limbu, Tai Le, Extended Latin, Extended Greek
- U+2xxx: Symbols (punctuation, super/subscripts, currency, letter-like, numerical, arrows, math, technical, OCR, boxes, dingbats, Braille), CJK radicals

Unicode Map
Basic Multilingual Plane

- U+3xxx: CJK symbols, Hiragana, Katakana, Bopomofo
- U+3400 to U+9FFF: CJK Unified Ideographs
- U+A000 to U+D7A3; Yi, Hangul Syllables
- U+D800 to U+DFFF; Surrogates *(no characters)*
- U+E000 to U+F8FF; Private Use
- U+Fxxx: CJK Compatibility Ideographs, Presentation Forms, Halfwidth/Fullwidth

Unicode Map
"Astral Planes"

- U+1xxxx: Archaic scripts (Linear B, Old Italic, Gothic, Ugaritic, Deseret, Shavian, Osmanya; more to come), math alphabets, music symbols (Western and Byzantine)
- U+2xxxx: Ultra-rare and specialized CJK ideographs
- U+30000 to U+DFFFF: Reserved
- U+Exxxx: Tag characters *(not for XML)*
- U+Fxxxx and U+10xxxx: Private Use

ENCODINGS

Pre-Unicode

- ASCII is a 7-bit encoding for about 100 characters
- ISO-8859-1 is an 8-bit encoding for about 200 characters
- Shift-JIS is a mixed 8/16-bit encoding for about 8,000 characters
- How to best encode Unicode's 1,114,112 possible codepoints?

Three Unicode Encodings

- UTF-16: 16-bit code units
- UTF-8: 8-bit code units
- UTF-32: 32-bit code units
- All have equal representation power
- All have advantages and disadvantages

UTF-16

- Each BMP character is represented by the obvious 16-bit code unit
- Other characters are represented by two consecutive 16-bit code units
- "A" is 0041
- Alpha is 0391
- Gothic Ahsa (U+10330) is D800 DB30

UTF-16 Byte Ordering

- By default, Unicode uses big-endian
- This can be overridden by local conventions (e.g. on Windows)
- U+FEFF, the Byte Order Mark or BOM, can be placed at the beginning of a file to unambiguously indicate the byte order, as U+FFFE does not exist

UTF-8

- Uses 1, 2, 3, or 4 bytes to encode a character
- No byte-ordering issue
- "A" is 41 (same as ASCII!)
- Alpha is CE 91
- Katakana "A" is E3 82 A2
- Gothic Ahsa is F0 90 8C B0

UTF-8 BOM

- UTF-8 does not need a BOM to determine byte order
- BOM byte sequence (EF BB BF) may still be useful in auto-detecting UTF-8
- Windows 2K and XP Notepad always generates it

UTF-32

- Encode each Unicode code point directly as 4 bytes
- Same byte ordering issues as UTF-16

Advantages of UTF-16

- *Almost* fixed-width encoding (non-BMP characters are expected to be rare in most documents)
- As compact as national CJK encodings (UTF-8 costs 50% more)
- Good compromise between space and ease of use

Advantages of UTF-8

- Fully ASCII-compatible, including control characters (but not Latin-1 compatible)
- First byte of any character indicates the number of trailing bytes to follow
- Sortable, searchable, compressible with 8-bit algorithms

Advantages of UTF-32

- Guaranteed fixed-width encoding
- Suitable for internal rather than external (file or network) use

SCSU

- Not a UTF, but a compression method
- ASCII-compatible (but not ASCII-control-character compatible)
- Universal decoding, source-specific encoding
- Uses about the same space as native 8-bit or 16-bit encodings

BOCU-1

- A different compression method
- Not compatible with anything else
- Universal decoding and encoding
- Uses about the same space as native 8-bit or 16-bit encodings

XML and Unicode

Larry Wall says:

"*An XML document knows what encoding it's in.*"

Choices, Choices...

- In UTF-8
- In UTF-16
- Something else
- All XML processors *required* to handle UTF-8 and UTF-16
- Most of them also handle at least ASCII and ISO-8859-1

UTF-8

- Given no other information, an XML document *must* be in UTF-8
- ASCII text is also UTF-8 text, so pure ASCII XML docs are fine as is
- *á* & friends aren't ASCII, though
- Variant approach: use ASCII, plus character references for everything else: for example, *á* is a

UTF-16

- Requires either a Byte Order Mark (which is not considered part of the XML document)...
- … or else an encoding declaration:

```
<?xml version="1.0"
    encoding="UTF-16"?>
```

Declare It Yourself

- Start the document with an encoding declaration
- This lets the processor figure out what's going on:

```
<?xml version="1.0"
      encoding="ISO-8859-1"?>
```

Deus Ex Machina

- Tell the processor what the encoding is outside the document
- Most common way is with a Content-Type: header
- Takes precedence over any encoding declaration within the XML document (but don't rely on this!)

Higher Levels of XML

- Higher levels don't really care how you do encoding
- Remember that character references are always Unicode code points
- A is "A"
- Α is Alpha
- 𐌰 is Gothic Ahsa

Early Uniform Normalization

- On the Web, document creators must *normalize* text (including HTML, XML) to avoid multiple spellings, signature issues
- Text in non-Unicode encodings is typically already normalized
- Details still being finalized

XML Names

- XML names (element type names, attribute names, enumerated attribute values, processing instruction targets, notation names) are based on Unicode 2.0 identifiers
- Generally, the first character must be a letter; others may be letters or digits
- *Any* character can appear in content

XML 1.1

- Extends XML names to make use of the full Unicode repertoire, except for defined exceptions
- Relies on the document author to choose sensible names.
- Adds NEL (U+0085) as a line end for IBM mainframe compatibility

THE PROGRAMMER'S VIEW

C or C++

- If you use UTF-8, you'll be able to use **`strcmp()`** and **`strlen()`** and so on
- If you use **`wchar_t`** (or MSTR in Visual Studio) you'll be able to use UTF-16
- Popular XML processors will give you either

Java

- The **`char`** type is 16 bits and pretty well forces UTF-16 down your throat internally
- Java can convert to and from almost anything externally
- All XML processors give you native UTF-16 strings

A Java Gotcha

- Avoid the Java methods `DataInputStream.readUTF` and `DataOutputStream.writeUTF;` they are only for binary string I/O
- Instead, create `InputStreamReader` or `OutputStreamWriter` objects with UTF-8 encodings

JavaScript

- (including JScript, ECMAScript, etc.)
- Strings are UTF-16 internally
- I/O is outside the scope of the language

Perl

- It just tries to do the right thing (using UTF-8 internally)
- XML::Parser (using James Clark's Expat) reads several encodings, delivers UTF-8
- Getting better all the time

Mozilla

- Also uses the Expat parser
- Delivers UTF-16 internally

International Components for Unicode (ICU)

- An Open Source C/C++ library that "does it all"
- Java version supplements native library
- Under active development by IBM and the programmer community
- The gold standard for supporting internationalization

ICU Features

- All components multi-thread safe
- Full Unicode string manipulation
- Complete locale support: more than 170 locales
- Fast and flexible character set conversion
- Efficient data loading mechanism

ICU Features

- Hierarchical resource bundles with flexible data storage mechanism
- Extensive calendar and timezone support
- Date, time, currency, number and message formatting
- Locale-sensitive sorting
- Locale-sensitive text boundary detection

ICU Features

- Customizable transliteration interface
- Unicode text compression algorithm
- Fast and compliant Unicode Bidi algorithm
- Most up-to-date Unicode support (including normalization)
- All APIs support UTF-16

ICU/J Features

- Advanced text boundary detection
- Hebrew, Islamic, Japanese, Thai calendar support
- Spelled-out numbers
- Normalization, transliteration, Unicode compression

Think Strings, Not Characters

- APIs should be designed around strings, not characters.
- Transformations often produce more or fewer characters in the output than in the input (uppercase ß is SS)
- Context is often critical: are we at a line or word boundary?

What is a String?

- In the C culture, a string is a bunch of bytes delimited by a 00 byte
- That means UTF-16 sequences are not really strings to C libraries
- UTF-8 shines here, because it is culturally compatible with C strings

What is a String?

- In higher-level languages, a string is an object: the internal representation can be hidden
- But it is important to note how the string indexes itself:
  - actual characters?
  - UTF-16 codepoints?

Sets and Tables

- Many character algorithms require tables indexed by a character
- An important special case: a set of Unicode characters (equivalent to a table with values 0 and 1)
- For 7-bit or 8-bit sets, a simple array is reasonable

Two-level Tables

- Many rows (256 codepoints) have similar or identical properties
- Entries in a global table can be a single value or point to a shared 256-entry sub-table
- Most entries tend to stay paged out
- Can also use 64-entry sub-tables

Inversion List

- Storing a set of Unicode characters as a list of integers
- Odd entries give starts, even entries give ends of ranges
- Binary search quickly determines membership
- Union, intersection, negation are fast and easy

Latin Letters Inversion List (only 20 entries)

```
0041 005B
0061 007B
00C0 00D7
00D8 00F7
00F8 0220
0222 0234
1E00 1E9C
1EA0 1EFA
FF21 FF3B
FF41 FF5B
```

SSGO

- Used internally by Mozilla
- Binary search through 6-byte Start, Size, Gap, Offset (for mapping) records
- Gap is 1 if every other codepoint belongs to the set
- Optimizations: fast-paths ASCII, skips unwanted blocks, provides cache

Storing Basic Unicode Properties in 32 Bits

- 5 bits for General Category
- 4 bits for bidirectional category
- 1 bit for bidirectional mirroring
- Remaining bits are category-sensitive:
  - Combining category for marks
  - Numeric value for numbers
  - Offset to opposite case for letters

Fast-pathing the BMP

- Almost all characters in running text will be in the BMP, with rare exceptions (text entirely in an archaic script, e.g.)
- It is worthwhile to optimize for the 16-bit case, especially in UTF-16 contexts
- Most BMP characters are below the surrogate range

Fast-pathing ASCII

- It is worthwhile to optimize for the ASCII case, especially in UTF-8 environments
- If most characters are ASCII, treat them in the main loop and special-case everything else

Ternary Search Trees

- Store long Unicode strings in tables without hashing
- Compromise between binary trees (space-efficient) and tries (time-efficient)
- Handle "don't care" matching smoothly

Culturally Correct Sorting

- Unicode binary code point order will not produce good results!
- International standards require at least a 3-level algorithm:
  - basic letters (*not* in codepoint order)
  - diacritics
  - upper vs. lower case

Sort Tailoring

- Different cultures have different rules
- Sorting rules depend on the user, not the source of data (Swedish names should be sorted English-style for an English user, not Swedish-style)
- ICU and other libraries have tailoring rules to support culture-specific rules

Matching, Indexing, Selecting

- The same rules apply as for sorting
- Matching may not be usable if it is strict; when matching directly from the user, allow for missing diacritics and other things

POINTS TO REMEMBER

Points to Remember

- If you have to internationalize, this is a good reason to choose XML
- If you have to use XML, this is a good reason to internationalize
- Paying for internationalization now is *much* cheaper than doing it later

More Information

http://www.unicode.org

http://www.ccil.org/~cowan/uamb.{ppt,sxi,pdf}

http://www.ccil.org/~cowan/uamb.html