The Trainer's Friend
INCORPORATED

An Introduction to Unicode

- ♦ Unicode Concepts and Terminology
- ♦ Unicode Mappings
- ♦ Appendix: UTF-32 Character Assignment Ranges
- ♦ Appendix: UTF-32 <-> UTF-16 <-> UTF-8 sample mappings

by Steve Comstock

The Trainer's Friend, Inc.
http://www.trainersfriend.com
303-393-8716
steve@trainersfriend.com

v1.7

Section Preview

❒ Introduction to Unicode

- ♦ Characters
- ♦ Characters, Glyphs, and Fonts
- ♦ Coding Schemes
- ♦ Code pages
- ♦ Standards
- ♦ Unicode

❒ Processing Unicode Characters

Characters

❒ **A <u>character</u> is an abstract concept that has evolved along with our understanding of language and information**

❒ **Initially, when most of us think of characters, we think of a particular character set: alphabetic letters, numeric characters, special characters, and so on**

- ♦ **But subtleties start to appear when you consider issues of other languages, different fonts, and the need for special meaning characters**

    - ✗ For example, you need characters to control a communications session when transmitting data

    - ✗ Obviously, you cannot use characters from the set of data you are sending to also be control characters: the communications process could not distinguish between data and the control functions

Characters, Glyphs, and Fonts

❒ **In computer terms, a character is <u>a grouping of bits</u> (binary ones and zeros) in packages of 8: one or more bytes**

❒ **There are two broad classes of characters: data characters and control characters**

- ♦ **Although one could make a case that data characters are just control characters whose function is to display a <u>glyph</u>**

  - ✗ A glyph is the visible representation of a character

- ♦ **Consider the [data] character called "upper case A"; the following are various glyphs that represent that character:**

  - ✗ A - Arial
  - ✗ A - Times New Roman
  - ✗ A - Courier new
  - ✗ A - Garamond
  - ✗ **A** - Bodoni
  - ✗ *A* - Park Avenue
  - ✗ ***A*** - FlamencoD

- ♦ **And so on; notice the concept of a <u>font</u> sneaking in here: a font is a set of glyphs used to represent a collection of characters [usually in a similar style]**

Coding Schemes

❐ But an upper case A is an upper case A, regardless of the glyph used to represent it

❐ In computers, we assign characters to bit patterns (and vice versa), and a "character" is an abstract thing, independent of any glyph

❐ The rules we use to make these assignments between characters and bit patterns are called coding schemes, and there are many in use today, for historical reasons

❐ There are three coding schemes most people in the IS industry need to be cognizant of:

- ♦ EBCDIC - Extended Binary Coded Decimal Interchange Code; used by IBM mainframes and AS/400 machines
- ♦ ASCII - American Standard Code for Information Interchange; used by almost all other hardware
- ♦ Unicode - gaining wide acceptance in use by software

❐ There are other coding schemes available, but from a practical point of view, we can get the vast majority, if not all, of our work done if we are aware of these coding schemes

Codepages

❒ **Even awareness of coding schemes is not quite enough to get us all we need for practical use**

❒ **Again, for historical and cultural reasons, many coding schemes have several variations, each slightly different than the others**

- ♦ **For example, in some environments you have need of a symbol like Å, but in other environments, users are not even aware of this character**

- ♦ **So computer designers introduced the concept of a codepage, which is a variation of a coding scheme**

❒ **After all, in 8 bits you only have 256 possible patterns**

- ♦ **You can run out of available characters pretty quick if you allow all those strange foreign, mathematical, scientific, engineering, currency, and other symbols**

❒ **The solution was to use codepages (**also spelled as two words: "code page" or "code pages"**)**

- ♦ **Users could set codepages for different environments**

- ♦ **Although you cannot mix codepages in a single environment: at any point in time your 256 bit patterns map to exactly one set of characters**

Codepages, continued

❐ **When a data character arrives in a computer from a magnetic tape, diskette, CD-ROM, network transmission, or so on, the computer just stores the character as it comes in, no judgements being made**

❐ **But consider what happens when a character comes in from a keyboard:**

- **The user presses a key with a glyph on it of, say, an upper case A**

- **The keyboard electronics assign a bit pattern to the character and transmit it to the computer, where it is received as part of an I/O program**

- **This I/O program may reassign the bit pattern before it is stored, depending on the current codepage:**

keyboard bit pattern **—> codepage —>** stored bit pattern

❐ **Similarly, when a character is sent by the computer to a printer or display unit, that output device has a codepage mapping followed by a font mapping to decide how to display the character on the device**

stored bit pattern **—> codepage —>** character **—> font —>** glyph

Standards

❒ To ensure consistency and clarity, a number of standards bodies have been created to develop and enhance standards for a variety of areas, including IS; these bodies include:

- ♦ ISO - the International Organization for Standardization
- ♦ ANSI - the American National Standards Institute, is the US member of ISO
- ♦ The ISO has a standard called ISCII which is very close to the ASCII character encoding standard

❒ Ultimately, one wants a single code page, a single, universal, encoding scheme for all characters

❒ From the perspective of international communications, one needs an encoding scheme that is

- ♦ Universal - covers all characters needed in all likely situations
- ♦ Efficient - avoids escape character sequences for special encoding, for example
- ♦ Unambiguous - every character has one and only one bit pattern mapping

Unicode

❒ Unicode is an encoding scheme developed by the Unicode Consortium (incorporated under the name Unicode, Inc. in 1991) and the ISO

❒ The Unicode Consortium is backed by most of the major players in the IS game, including these (and many more):

- ♦ Adobe Systems
- ♦ Apple Computer
- ♦ Compaq Computer
- ♦ Ericsson Mobile Communications
- ♦ Hewlett-Packard
- ♦ IBM
- ♦ Microsoft
- ♦ NCR
- ♦ Netscape
- ♦ Oracle
- ♦ PeopleSoft
- ♦ Quark
- ♦ SAP
- ♦ SAS Institute
- ♦ SHARE
- ♦ Software AG
- ♦ Sun Microsystems
- ♦ Sybase
- ♦ Unisys

Unicode, continued

❒ **In 1992, the Unicode Consortium and ISO agreed to merge their character encoding standard, so the character sets map exactly**

- ♦ **In addition to assigning names and bit-pattern mappings to characters, in conjunction with the ISO, the Unicode standard also provides implementation algorithms, properties, and semantic information**

❒ **The basic, original premise, was to use 16-bits for every character**

- ♦ **This allows for 64K unique patterns (65,536)**

- ♦ **Maintaining compatibility with as many already existing standards as possible**

❒ **By the year 2000, however, it was clear that more character space was needed**

- ♦ **In May of 2001, 44,946 new characters were added (mostly CJK (Chinese, Japanese, Korean) characters, along with some historic scripts and several sets of symbols)**

  - ✗ As of Unicode standard 3.1 there were 94,140 characters included

- ♦ **As of Unicode standard 4.0 (June, 2003), there are 96,382 characters in the standard, and for 4.1 (March, 2005) the count is now 97,655; for 5.0 (July, 2006), there are 99,024**

Unicode, continued

❒ **There are three alternative ways of representing Unicode characters, including:**

- ♦ **UTF-16 — the basic 16-bit encoding scheme: two bytes used for every Unicode character**

  - ✗ But version 3.0 of the Unicode standard introduced a concept called surrogate pairs that allows some Unicode characters to be represented by a pair of two-byte values

- ♦ **UTF-8 — an algorithm for converting Unicode characters to a string of characters that are one, two, three, or four bytes in length, and back**

- ♦ **UTF-32 — a 32-bit encoding, the basis for the ultimate character encoding, allowing for 1,114,112 character assignments (note: the leftmost 11 of the 32 bits must be all binary zeros)**

  - ✗ This encoding was made an official part of the Unicode standard in version 3.1 in May, 2001

❒ **UTF stands for Unicode Transformation Format**

❒ **Here are some pointers to Web sites for more information about Unicode:**

- ♦ **Unicode home page: http://www.unicode.org**

- ♦ **IBM: http://www-106.ibm.com/developerworks/unicode/**

Unicode, continued

- ❒ **So why do we care about this on the mainframe?**
  - ♦ **IBM is trying to position mainframes as the ultimate server for intranets and the Internet / World Wide Web**
    - ✗ Web pages are generally coded in HTML (HyperText Markup Language) or XHTML (eXtensible HyperText Markup Language)
      - ➢ HTML 4 and all versions of XHTML require support for Unicode
  - ♦ **XML (eXtensible Markup Language) is becoming one of the premier data exchange formats - requires Unicode support**
  - ♦ **At some point in time, ("not too far down the road" to quote one of the z/OS developers) z/OS will require the Unicode support functions be installed**
  - ♦ **DB2 can store / access Unicode data in CHAR, VARCHAR, and CLOB data types**
    - ✗ In Version 8, the DB2 catalog is stored in UTF-8
  - ♦ **Many databases and programming languages on UNIX and Windows machines support Unicode**
  - ♦ **Current mainframe compilers (COBOL, PL/I, C, C++) all support Unicode**
- ❒ **Unicode can provide, eventually, the ability to have a single codepage yet support all languages simultaneously**

Unicode, concluded

❒ **Although there are people who are against Unicode (and even some competing standards), Unicode seems to be the way of the future**

- ♦ **Enabling single encoding and data interchange across platforms and simultaneous multiple language support on screens and reports**

❒ **Also note that z-series machines have a number of instructions that work with Unicode, in the UTF-16 format (PKU, UNPKU, CLCLU, MVCLU, TROO, TROT, TRTO, TRTT)**

- ♦ **But UTF-8 seems to be the most widely used format on the Web and, probably, in XML that's not even used on the Web**
- ♦ **Instructions to convert between UTF-16 and UTF-8 have been available since machines introduced in 1999 (CUTFU, CUUTF)**
- ♦ **In 2004, instructions were added to convert between:**

✗	UTF-8	<-->	UTF-16	(new names for old instructions)
✗	UTF-8	<-->	UTF-32	(new instructions)
✗	UTF-16	<-->	UTF-32	(new instructions)

Section Preview

❒ **Processing Unicode Characters**

- ♦ **Unicode Representations**
- ♦ **UTF-32: Unicode Scalar Values**
- ♦ **UTF-16**
- ♦ **Surrogate Pairs**
- ♦ **UTF-16 -> Unicode Scalar Value**
- ♦ **UTF-32 -> UTF-16**
- ♦ **UTF-8**
- ♦ **UTF-32 -> UTF-8**
- ♦ **. UTF-8 -> UTF-32**
- ♦ **. Other Mappings**
- ♦ **. Unicode - Conclusion**

Unicode Representations

❒ **This section is a techincal discussion of how Unicode characters are stored using the three formats: UTF-8, UTF-16, and UTF-32**

- ♦ **According to the standard, these are considered equally valid**
  - ✗ In the sense that every Unicode character may be represented in any of these formats, and the mapping between formats is well-defined
- ♦ **To work with Unicode data, one has to know which eoncoding format has been used**
  - ✗ This may be supplied external to the data itself, as in an HTTP header or HTML META statement, for example
  - ✗ In some cases, the program processing a string may be able to examine the string and deduce the format being used (but this is not preferred)

UTF-32: Unicode Scalar Values

- ❐ **Every Unicode character is assigned an integer value, the** Unicode scalar value

- ❐ **UTF-32** is **the set of Unicode Scalar Values**

  - ♦ **This is also sometimes called UCS-4, meaning the Universal Character Set as 4-bytes per character**

- ❐ **The possible range of values in the Unicode scalar set is `x'00000000'` - `x'0010FFFF'` or, in binary, the maximum allowed value is**

```
0000 0000 0001 0000 1111 1111 1111 1111
```

  - ♦ **21 bits; in decimal, the values range from 0 to 1,114,111**

  - ♦ **Every Unicode character is assigned a number in this range, a point along the string of integers in this range (this is sometimes called a** code point**)**

    - ✗ Although not every number in this range is assigned to a Unicode character

    - ✗ Also, a subset of this range is reserved for surrogate pairs...

UTF-16

❒ UTF-16 was the beginning point of Unicode character assignments

❒ Initially, each UTF-16 character was a single two-byte unit

- ♦ But when surrogates needed to be introduced, to accomodate a larger character set, some characters became represented by a single two-byte unit, others by a pair of two-byte units

❒ When a processing program such as a browser or editor is working with UTF-16 data, it assumes each two-byte unit represents a character

- ♦ Except that certain values are reserved to represent surrogate pairs: situations where it takes two two-byte units to represent a single character

❒ The theoretical range of Unicode scalar values is `x'0000 0000'` - `x'0010 FFFF'`

- ♦ For Unicode scalar values greater than or equal to `x'0001 0000'`, surrogate pairs are used

- ♦ Values in the range `x'0000 D800'` - `x'0000 DFFF'` are reserved for use in surrogate pairs

Surrogate Pairs

❐ **How to recognize when a two-byte unit begins a surrogate pair?**

- **If a UTF-16 unit has a value in the range `x'D800'` - `x'DBFF'`, that unit is a high surrogate and you need to combine that two-byte unit and the next, which must be a low surrogate, to determine the actual character that is being represented**

- **Low surrogates are in the range `x'DC00'` - `x'DFFF'`**

    ✗ It is an error for a low surrogate not to be preceded by a high surrogate, and for a high surrogate not to be followed by a low surrogate

<u>In binary:</u>

high surrogates:	**1101 1000 0000 0000 - 1101 1011 1111 1111**
low surrogates:	**1101 1100 0000 0000 - 1101 1111 1111 1111**

<u>In decimal:</u>

high surrogates:	**55,296 - 56,319**
low surrogates:	**56,320 - 57,343**

<u>Notes</u>

- **Each surrogate range contains 1,024 values, so the possible number of surrogate pair values is 1,024 x 1,024 or 1,048,576**

- **The vast majority of characters do not require surrogate pairs**

UTF-16 -> Unicode Scalar Value

- **The algorithm to convert from a UTF-16 Unicode character to the Unicode scalar value (in other words, UTF-16 -> UTF-32) is this:**

  - **If a two-byte unit is not a surrogate value, the Unicode scalar value is the two-byte value itself**

    - So if the two- byte unit is in the range `x'0000'`- `x'D7FF'` or `x'E000'` - `x'FFFF'`, the Unicode scalar value is that value (or, equivalently, `x'0000 0000'` - `x'0000 D7FF'` and `x'0000 E000'` - `x'0000 FFFF'`)

  - **If a two-byte unit is a surrogate value, the character is composed of two two-byte units, so calculate the Unicode scalar value as** the sum of

    - (The high surrogate - **`x'D800'`**) * **`x'0400'`**

    - (The low surrogate - **`x'DC00'`**)

    - **`x'0001 0000'`**

- **We examine this algorithm more carefully ...**

UTF-16 -> Unicode Scalar Value, continued

Notes

- **The first calculation provides the displacement into the high surrogate range (resulting in a number in the range x'0000'- x'03FF' or, in decimal, 0 - 1023 or, in binary: b'0000 0000 0000 0000'- b'0000 0111 1111 1111')**

- **Multiplying by x'0400' (decimal 1024) effectively shifts the value to the left 10 bits, producing numbers in the range x'0000 0000'- x'000 FFC00' with the last 10 bits all zeros**

```
         0000 00xx xxxx xxxx

0000 0000 xxxx xxxx xx00 0000 0000
```

- **The second value is the displacement into the low surrogate range (also resulting in a number in the range x'0000'- x'03FF' or, in decimal, 0 - 1023)**

- **Adding the two numbers (inserting leading zeros in the first to make them the same length) and the x'1 0000':**

```
    0000 0000 0000 xxxx xxxx xx00 0000 0000
    0000 0000 0000 0000 0000 00yy yyyy yyyy
+   0000 0000 0000 0001 0000 0000 0000 0000
```

- **Adding the x'0001 0000' ensures the resulting Unicode scalar values are in the range x'0001 0000'- x'0010 FFFF'**

UTF-16 -> Unicode Scalar Value, continued

- **We can look at Unicode scalar assignments this way:**

`x'0000 0000' - x'0000 D7FF'`	basic 16-bit codes
`x'0000 D800' - x'0000 DFFF'`	surrogate values (assigned, but not to characters)
`x'0000 E000' - x'0000 FFFF'`	basic 16-bit codes
`x'0001 0000' - x'0010 FFFF'`	computed from surrogate pairs

- **These ranges may be further subdivided for study but these subsets are not of interest in this paper**
  - **However, the next level of detail is presented in the first Appendix to this document**
  - **The second Appendix lists specific mapping values between UTF-32, UTF-16, and UTF-8**

- **At one point in the cycle of development, there was a Unicode encoding called UCS-2 (Universal Character Set as 2-bytes per character)**
  - **UCS-2 is, essentially, UTF-16 without support for surrogate pairs**
    - ✗ UCS-2 is currently supplanted by UTF-16

UTF-32 -> UTF-16

- ❐ **On the other hand, given a UTF-32 character, how do you represent it in UTF-16?**

  - ♦ **This algorithm, of course, reverses the steps before:**

    - ✗ For characters less than `x'0001 0000'`, the UTF-16 representation is simply the rightmost 16 bits

    - ✗ For characters in the range `x'0001 0000'` to `x'0010 FFFF'` we need to build the high surrogate (HS) and low surrogate (LS) two-byte patterns, as follows...

      - ➢ Subtract `x'0001 0000'`; this gives a value in the range `x'0000 0000'` to `x'000F FFFF'`, call this value **char**

      - ➢ HS = `x'D800'` + (**char** / `x'400'`)
      - ➢ LS = `x'DC00'` + (**char** % `x'400'`)

    - ✗ In other words, consider the 20 rightmost bits of **char** to be designated this way:

```
xxxx xxxx xx|yy yyyy yyyy
```

    - ✗ then the HS is `x'D800'` plus the leftmost 10 bits (the `x's`) and the LS is `x'DC00'` plus the rightmost 10 bits (the `y's`)

UTF-8

❒ **One of the major reasons Unicode has been successful is a deliberate decision to encompass as many existing standards as possible**

❒ **When it came to the 7-bit ASCII / ISCII standard, Unicode allowed an 8-bit representation for characters with binary code points from b'0000 0000' to b'0111 1111'**

- ♦ **In other words, the first 127 Unicode characters are ASCII!**

❒ **UTF-8 allows you to represent any Unicode character as a string of one, two, three, or four bytes!**

❒ **Conversely, when a processor knows it is dealing with UTF-8, it starts by looking at one byte and the value will imply whether it needs to use one, two, three, or four bytes to construct the UTF-32 value (Unicode scalar value)**

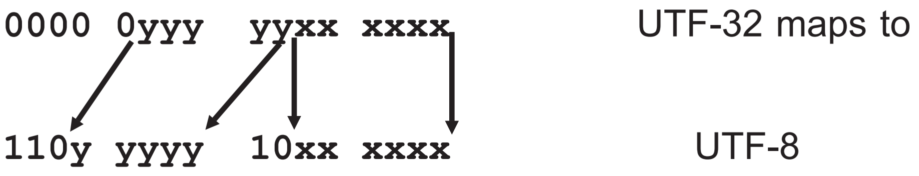
UTF-32 -> UTF-8

❐ **The mapping works like this**

- **Unicode scalar values in the range 00000000-0000007F map to single byte values 00-7F**

- **Unicode scalar values in the range 00000080-000007FF map to two byte values, where the first byte is in the range C2-DF and the second byte is in the range 80-BF**

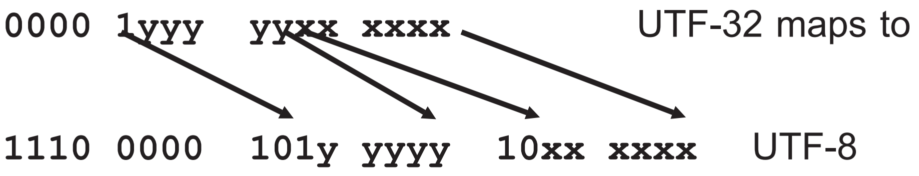
  ✗ Specifically looking at the bit patterns:



```
0000 0yyy  yyxx xxxx          UTF-32 maps to

110y yyyy  10xx xxxx             UTF-8
```

- **Unicode scalar values in the range 00000800-00000FFF map to three byte values, where the first byte is E0, the second byte is in the range A0-BF, and the third byte is in the range 80-BF**

  ✗ Specifically looking at the bit patterns:



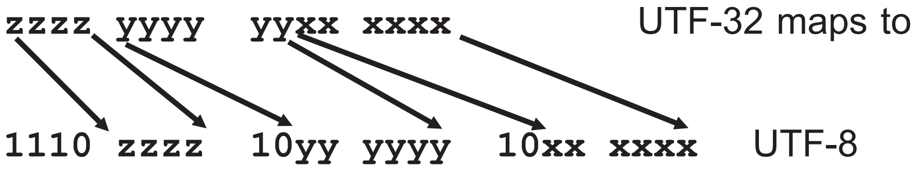
```
0000 1yyy  yyxx xxxx          UTF-32 maps to

1110 0000  101y yyyy  10xx xxxx   UTF-8
```

UTF-32 -> UTF-8, continued

❒ **The mapping works like this, continued**

- ♦ **Unicode scalar values in the range 00001000-0000FFFF map to three byte values, where the first byte is in the range E1-EF, the second byte is in the range 80-BF, and the third byte is in the range 80-BF**

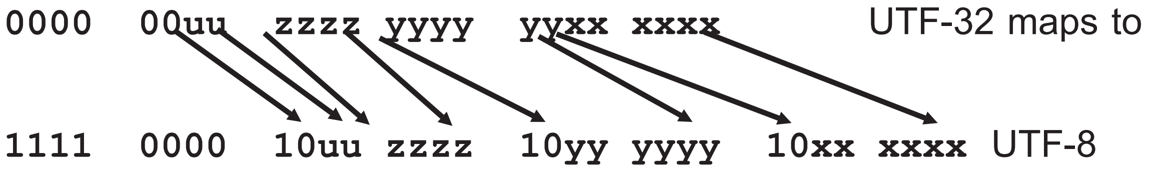
  ✗ Specifically looking at the bit patterns:



```
zzzz yyyy  yyxx xxxx          UTF-32 maps to

1110 zzzz  10yy yyyy  10xx xxxx   UTF-8
```

- ♦ **Unicode scalar values in the range 00010000-0003FFFF map to four byte values, where the first byte is F0, the second byte is in the range 90-BF, the third byte is in the range 80-BF, and the fourth byte is in the range 80-BF**

  ✗ Specifically looking at the bit patterns:



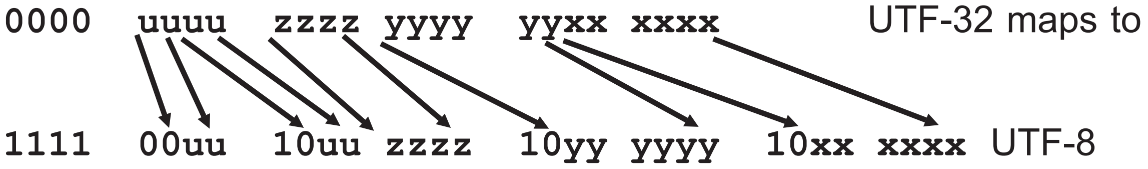
```
0000  00uu  zzzz yyyy  yyxx xxxx        UTF-32 maps to

1111  0000  10uu zzzz  10yy yyyy  10xx xxxx UTF-8
```

UTF-32 -> UTF-8, continued

- **The mapping works like this, continued**

  - **Unicode scalar values in the range 00040000-000FFFFF map to four byte values, where the first byte is in the range F1-F3, the second byte is in the range 80-BF, the third byte is in the range 80-BF, and the fourth byte is in the range 80-BF**

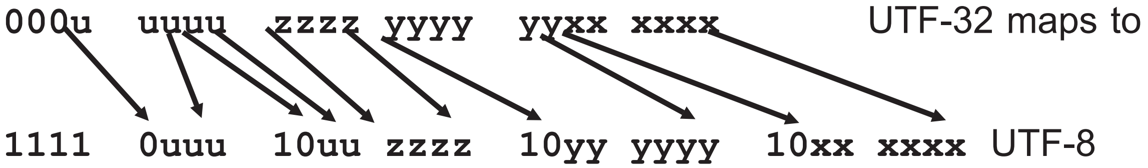
    - Specifically looking at the bit patterns:



```
0000  uuuu  zzzz yyyy  yyxx xxxx          UTF-32 maps to

1111  00uu  10uu zzzz  10yy yyyy  10xx xxxx UTF-8
```

  - **Unicode scalar values in the range 00100000-0010FFFF map to four byte values, where the first byte is F4, the second byte is in the range 80-BF, the third byte is in the range 80-BF, and the fourth byte is in the range 80-BF**

    - Specifically looking at the bit patterns:



```
000u  uuuu  zzzz yyyy  yyxx xxxx          UTF-32 maps to

1111  0uuu  10uu zzzz  10yy yyyy  10xx xxxx UTF-8
```

UTF-8 -> UTF-32

❒ Obviously, the reverse mapping works based on the value a processor finds in the first byte of a UTF-8 string:

- ♦ If a byte is in the range 00-7F, it constructs the UTF-32 output by prefixing hex 000000
- ♦ If a byte is in the range C2-DF, it knows to take that byte and the next to build the UTF-32 value
- ♦ If a byte is E0-EF, it knows to take that byte and the next two bytes to build the UTF-32 value
- ♦ If a byte is F0-F4, it knows to take that byte and the next three to build the UTF-32 scalar value and then to map that to the surrogate pair
- ♦ Any other values in the first byte indicate an error

❒ The mechanics should be reasonably apparent from the preceding pages, and details are left as a task for the interested reader to do on your own

- ♦ The second Appendix to this document shows many code point mappings between UFT-32, UTF-16, and UTF-8

Other Mappings

❐ We have not discussed these mappings

- ♦ UTF-8 -> UTF-16
- ♦ UTF-16 -> UTF-8

❐ You can accomplish these mappings by using UTF-32 as an intermediate format then using the existing UTF-32 <-> UTF-16 mappings

- ♦ Or it's possible to devise direct mappings based on what we've discussed

❐ On IBM z/Architecture machines, there are instructions that convert between Unicode formats:

- ♦ CU12 - from UTF-8 to UTF-16 (also known as CUTFU)
- ♦ CU21 - from UTF-16 to UTF-8 (also knon as CUUTF)
- ♦ CU14 - from UTF-8 to UTF-32
- ♦ CU41 - from UTF-32 to UTF-8
- ♦ CU24 - from UTF-16 to UTF-32
- ♦ CU42 - from UTF-32 to UTF-16

❐ Our interest in this paper has been simply in demonstrating the interrelationships between the three Unicode formats and we feel that has been explored thoroughly enough at this point

Endian-ness

❒ **One issue we didn't raise that needs to be raised in certain circumstances: UTF-16 and UTF-32 each have variants depending on the order the bytes are physically stored**

- ♦ **In mainframe (and many other systems) these characters are stored in Big Endian (BE) format: most signficant byte first**

  - ✗ Officially called UTF-16BE and UTF-32BE

- ♦ **In other systems, characters are stored in Little Endian (LE) format: least signficant byte first**

  - ✗ Officially designated UTF-16LE and UTF-32LE

❒ **When passing UTF-16 and UTF-32 strings, one has to specify the "endian-ness" of the strings, in headers or by inserting a hex value called a Byte Order Mark (BOM) at the front of the data**

- ♦ **In UTF-16, an intial byte sequence of `x'FEFF'` indicates big endian encoding, while `x'FFFE'` indicates little endian encoding**

- ♦ **In UTF-32, an initial byte sequence of `x'0000 FEFF'` indicates big endian encoding, while `x'FFFE 0000'` indicates little endian order**

- ♦ **In both cases, any BOM is not considered part of the data, and the absence of any BOM value implies big endian encoding**

Unicode - Conclusion

❒ Unicode is here, is well-defined, and is gaining wide acceptance on a variety of hardware and software platforms

❒ Although new characters and additional refinements continue to be made, the Unicode Consortium strives to make each new version backward compatible

- ♦ In other words, current algorithms and mappings will carry forward

❒ There is plenty of work to do for those interested in exploring the internationalization of the Web and of computer work in general

- ♦ The ultimate goal is effective communication between men and women of all languages and cultures on the planet

Section Preview

❒ **Appendix A**

- ♦ **UTF-32 character allocation ranges**

UTF-32 Character Allocation Ranges

❒ **Every Unicode character is assigned a <u>scalar value</u>: an integer**

- ♦ **This is a 21-bit number in the range:**

binary
000000000000000000000 - 1 0000 1111 1111 1111 1111

or

hex	**decimal**
00 00 00 - 10 FF FF	**0 - 1,114,111**

UTF-32 Assignments, continued

- UTF-32 is really just the 21-bit numbers, right justified, padded on the left with 11 binary zeros (so only six relevant hex digits)

  - General allocation is this (not all gaps and details are shown):

00 00 00 - 00 00 7F	7-bit ASCII	
00 00 80 - 00 00 FF	Controls and Latin-1 Supplement	
00 01 00 - 00 01 7F	Latin Extended-A	
00 01 80 - 00 02 4F	Latin Extended-B	
00 02 50 - 00 02 AF	IPA extensions (some Latin and Greek)	
00 02 B0 - 00 02 FF	Spacing modifier letters	
00 03 00 - 00 03 6F	Combining diacritical marks	
00 03 70 - 00 03 FF	Greek and Coptic	
00 04 00 - 00 04 FF	Cyrillic	
00 05 00 - 00 05 2F	Cyrillic supplementary	
00 05 30 - 00 05 8F	Armenian	
00 05 90 - 00 05 FF	Hebrew	
00 06 00 - 00 06 FF	Arabic	
00 07 00 - 00 07 4F	Syriac	
00 07 50 - 00 07 7F	Thaana	(note gap here)
00 09 00 - 00 09 7F	Devanagari	
00 09 80 - 00 09 FF	Bengali	
00 0A 00 - 00 0A 7F	Gurmukhi	
00 0A 80 - 00 0A FF	Gujarati	
00 0B 00 - 00 0B 7F	Oriya	
00 0B 80 - 00 0B FF	Tamil	
00 0C 00 - 00 0C 7F	Telugu	
00 0C 80 - 00 0C FF	Kannada	
00 0D 00 - 00 0D 7F	Malayalam	
00 0D 80 - 00 0D FF	Sinhala	
00 0E 00 - 00 0E 7F	Thai	
00 0E 80 - 00 0E FF	Lao	
00 0F 00 - 00 0F FF	Tibetan	
00 10 00 - 00 10 9F	Myanamar	
00 10 A0 - 00 10 FF	Georgian	
00 11 00 - 00 11 FF	Hangul Jamo	

UTF-32 Assignments, continued

❒ UTF-32 scalar values allocation, continued:

00 12 00 - 00 13 7F	Ethiopic	
00 13 A0 - 00 13 FF	Cherokee	
00 14 00 - 00 16 7F	Unified Canadian Aboriginal symbols	
00 16 80 - 00 16 9F	Ogham	
00 16 A0 - 00 16 FF	Runic	
00 17 00 - 00 17 1F	Tagalog	
00 17 20 - 00 17 3F	Hanunoo	
00 17 40 - 00 17 5F	Buhid	
00 17 60 - 00 17 7F	Tagbanwa	
00 17 80 - 00 17 FF	Khmer	
00 18 00 - 00 18 AF	Mongolian	(note gap here)
00 19 00 - 00 19 4F	Limbu	
00 19 50 - 00 19 7F	Tai Le	(note gap here)
00 19 E0 - 00 19 FF	Khmer symbols	(note gap here)
00 1D 00 - 00 1D 7F	Phonetic extensions	(note gap here)
00 1E 00 - 00 1E FF	Latin extended additional	
00 1F 00 - 00 1F FF	Greek extended	
00 20 00 - 00 20 6F	General punctuation	
00 20 70 - 00 20 9F	Superscripts and subscripts	
00 20 A0 - 00 20 CF	Currency symbols	
00 20 D0 - 00 20 FF	Combining diacritical marks for symbols	
00 21 00 - 00 21 4F	Letterlike symbols	
00 21 50 - 00 21 8F	Number forms	
00 21 90 - 00 21 FF	Arrows	
00 22 00 - 00 22 FF	Mathematical operators	
00 23 00 - 00 23 FF	Miscellaneous technical	
00 24 00 - 00 24 3F	Control pictures	
00 24 40 - 00 24 5F	Optical character recognition	
00 24 60 - 00 24 FF	Enclosed alphanumerics	
00 25 00 - 00 25 7F	Box drawing	
00 25 80 - 00 25 9F	Block elements	
00 25 A0 - 00 25 FF	Geometric shapes	
00 26 00 - 00 26 FF	Miscellaneous symbols	

UTF-32 Assignments, continued

❒ UTF-32 scalar values allocation, continued:

00 27 00 - 00 27 BF	Dingbats	
00 27 C0 - 00 27 EF	Miscellaneous mathematical symbols-A	
00 27 F0 - 00 27 FF	Supplemental arrows-A	
00 28 00 - 00 28 FF	Braille patterns	
00 20 00 - 00 29 7F	Supplemental arrows-B	
00 29 80 - 00 29 FF	Miscellaneous mathematical symbols-B	
00 2A 00 - 00 2A FF	Supplemental mathematical operators	
00 2B 00 - 00 2B FF	Miscellaneous symbols and arrows	
00 2C 00 - 00 2E 7F	unassigned	
00 2E 80 - 00 2E FF	CJK radicals supplement	
00 2F 00 - 00 2F DF	Kangxi radicals	(note gap here)
00 2F F0 - 00 2F FF	Ideographic description characters	
00 30 00 - 00 30 3F	CJK symbols and punctuation	
00 30 40 - 00 30 9F	Hiragana	
00 30 A0 = 00 30 F0	Katakana	
00 31 00 - 00 31 2F	Bopomofo	
00 31 30 - 00 31 8F	Hangul compatibility Jamo	
00 31 90 - 00 31 9F	Kanbun	
00 31 A0 - 00 31 BF	Bopomofo extended	(note gap here)
00 31 F0 - 00 31 FF	Katakana phonetic extensions	
00 32 00 - 00 32 FF	Enclosed CJK letters and months	
00 33 00 - 00 33 FF	CJK compatibility	
00 34 00 - 00 4D BF	CJK unified ideographs extension A	
00 4D C0 - 00 4D FF	Yijing Hexagram symbols	
00 4E 00 - 00 9F AF	CJK unified ideographs	
00 A0 00 - 00 A4 8F	Yi syllables	
00 A4 90 - 00 A4 CF	Yi radicals	(note gap here)
00 AC 00 - 00 D7 AF	Hangul syllables	(note gap here)
00 D8 00 - 00 DB FF	high surrogates	
00 DC 00 - 00 DF FF	low surrogates	(note gap here)
00 F9 00 - 00 FA FF	CJK compatibility ideographs	
00 FB 00 - 00 FB 4F	Alphabetic presentation forms	
00 FB 50 - 00 FD FF	Arabic presentation forms-A	

UTF-32 Assignments, continued

❒ UTF-32 scalar values allocation, continued:

00 FE 00 - 00 FE 0F	Variation selectors	
00 FE 20 - 00 FE FF	Combining half marks	
00 FE 30 - 00 FE 4F	CJK compatibility forms	
00 FE 50 - 00 FE 6F	Small form variants	
00 FE 70 - 00 FE FF	Arabic presentation forms-B	
00 FF 00 - 00 FF EF	Halfwidths and fullwidth forms	
00 FF F0 - 00 FF FF	Specials	

---

01 00 00 - 01 00 7F	Linear B syllabary	
01 00 80 - 01 00 FF	Linear B ideograms	
01 01 00 - 01 01 3F	Aegean numbers	(note gap here)
01 03 00 - 01 03 2F	Old Italic	
01 03 30 - 01 03 4F	Gothic	(note gap here)
01 03 80 - 01 03 9F	Ugaritic	(note gap here)
01 04 00 - 01 04 4F	Deseret	
01 04 50 - 01 04 7F	Shavian	
01 04 80 - 01 04 AF	Osmanya	(note gap here)
01 08 00 - 01 08 3F	Cypriot syllabary	(note gap here)
01 D0 00 - 01 D0 FF	Byzantine musical symbols	
01 D1 00 - 01 D1 FF	Musical symbols	(note gap here)
01 D3 00 - 01 D3 5F	Tai Xuan Jing symbols	(note gap here)
01 D4 00 - 01 D7 FF	Mathematical alphanumeric symbols	
01 D8 00 - 01 FF FF	unassigned	
02 00 00 - 02 A6 DF	CJK unified ideographs extension B	
02 A6 E0 - 02 F7 FF	unassigned	
02 F8 00 - 02 FA 1F	CJK compatibility ideographs supplement	
02 FA 20 - 0D FF FF	unassigned	
0E 00 00 - 0E 00 7F	Tags	(note gap here)
0E 01 00 - 0E 01 EF	Variant selectors supplement	
0E 01 F0 - 0E FF FF	unassigned	
0F 00 00 - 0F FF FD	Private use area	
0F FF FE - 0F FF FF	non-characters	
10 00 00 - 10 FF FD	Private use area	
10 FF FE - 10 FF FF	non-characters	

This page intentionally left almost blank.

Section Preview

❒ **Appendix B**

- ♦ **UTF-32 <-> UTF-16 <-> UTF-8 sample mappings**

Mappings

UTF-32	UTF-16	UTF-8
00 00 00	00 00	00
00 00 01	00 01	01
.		
.		
.		
00 00 7E	00 7E	7E
00 00 7F	00 7F	7F
00 00 80	00 80	C2 80
00 00 81	00 81	C2 81
.		
.		
.		
00 00 BE	00 BE	C2 BE
00 00 BF	00 BF	C2 BF
00 00 C0	00 C0	C3 80
00 00 C1	00 C1	C3 81
.		
.		
.		
00 00 FE	00 FE	C3 BE
00 00 FF	00 FF	C3 BF
00 01 00	01 00	C4 80
00 01 01	01 01	C4 81
.		
.		
.		
00 01 3E	01 3E	C4 BE
00 01 3F	01 3F	C4 BF
00 01 40	01 40	C5 80
00 01 41	01 41	C5 81
.		
.		
.		

Mappings, 2

UTF-32	UTF-16	UTF-8
00 01 7E	01 7E	C5 BE
00 01 7F	01 7F	C5 BF
00 01 80	01 80	C6 80
00 01 81	01 81	C6 81
.		
.		
.		
00 01 BE	01 BE	C6 BE
00 01 BF	01 BF	C6 BF
00 01 C0	01 C0	C7 80
00 01 C1	01 C1	C7 81
.		
.		
.		
00 01 FE	01 FE	C7 BE
00 01 FF	01 FF	C7 BF
00 02 00	02 00	C8 80
00 02 01	02 01	C8 81
.		
.		
.		
00 02 3E	02 3E	C8 BE
00 02 3F	02 3F	C8 BF
00 02 40	02 40	C9 80
00 02 41	02 41	C9 81
.		
.		
.		
00 02 7E	02 BE	C9 BE
00 02 7F	02 BF	C9 BF
00 02 80	02 00	CA 80
00 02 81	02 01	CA 81
.		
.		
.		

Mappings, 3

UTF-32	UTF-16	UTF-8
00 02 BE	02 BE	CA BE
00 02 BF	02 BF	CA BF
00 02 C0	02 C0	CB 80
00 02 C1	02 C1	CB 81
.		
.		
.		
00 02 FE	02 FE	CB BE
00 02 FF	02 FF	CB BF
00 03 00	03 00	CC 80
00 03 01	03 01	CC 81
.		
.		
.		
00 03 3E	03 3E	CC BE
00 03 3F	03 3F	CC BF
00 03 40	03 40	CD 80
00 03 41	03 41	CD 81
.		
.		
.		
00 03 7E	03 7E	CD BE
00 03 7F	03 7F	CD BF
00 03 80	03 80	CE 80
00 03 81	03 81	CE 81
.		
.		
.		

Mappings, 4

UTF-32	UTF-16	UTF-8
00 03 BE	03 BE	CE BE
00 03 BF	03 BF	CE BF
00 03 C0	03 C0	CF 80
00 03 C1	03 C1	CF 81
.		
.		
.		
00 03 FE	03 FE	CF BE
00 03 FF	03 FF	CF BF
00 04 00	04 00	D0 80
00 04 01	04 01	D0 81
.		
.		
.		
00 04 3E	04 3E	D0 BE
00 04 3F	04 3F	D0 BF
00 04 40	04 40	D1 80
00 04 41	04 41	D1 81
.		
.		
.		
00 04 7E	04 7E	D1 BE
00 04 7F	04 7F	D1 BF
00 04 80	04 80	D2 80
00 04 81	04 81	D2 81
.		
.		
.		

Mappings, 5

UTF-32	UTF-16	UTF-8
00 04 BE	04 BE	D2 BE
00 04 BF	04 BF	D2 BF
00 04 C0	04 C0	D3 80
00 04 C1	04 C1	D3 81
.		
.		
.		
00 04 FE	04 FE	D3 BE
00 04 FF	04 FF	D3 BF
00 05 00	05 00	D4 80
00 05 01	05 01	D4 81
.		
.		
.		
00 05 3E	05 3E	D4 BE
00 05 3F	05 3F	D4 BF
00 05 40	05 40	D5 80
00 05 41	05 41	D5 81
.		
.		
.		
00 05 7E	05 7E	D5 BE
00 05 7F	05 7F	D5 BF
00 05 80	05 80	D6 80
00 05 81	05 81	D6 81
.		
.		
.		

Mappings, 6

UTF-32	UTF-16	UTF-8
00 05 BE	05 BE	D6 BE
00 05 BF	05 BF	D6 BF
00 05 C0	05 C0	D7 80
00 05 C1	05 C1	D7 81
.		
.		
.		
00 05 FE	05 FE	D7 BE
00 05 FF	05 FF	D7 BF
00 06 00	06 00	D8 80
00 06 01	06 01	D8 81
.		
.		
.		
00 06 3E	06 3E	D8 BE
00 06 3F	06 3F	D8 BF
00 06 40	06 40	D9 80
00 06 41	06 41	D9 81
.		
.		
.		
00 06 7E	06 7E	D9 BE
00 06 7F	06 7F	D9 BF
00 06 80	06 80	DA 80
00 06 81	06 81	DA 81
.		
.		
.		

Mappings, 7

UTF-32	UTF-16	UTF-8
00 06 BE	06 BE	DA BE
00 06 BF	06 BF	DA BF
00 06 C0	06 C0	DB 80
00 06 C1	06 C1	DB 81
.		
.		
.		
00 06 FE	06 FE	DB BE
00 06 FF	06 FF	DB BF
00 07 00	07 00	DC 80
00 07 01	07 01	DC 81
.		
.		
.		
00 07 3E	07 3E	DC BE
00 07 3F	07 3F	DC BF
00 07 40	07 40	DD 80
00 07 41	07 41	DD 81
.		
.		
.		
00 07 7E	07 7E	DD BE
00 07 7F	07 7F	DD BF
00 07 80	07 80	DE 80
00 07 81	07 81	DE 81
.		
.		
.		

Mappings, 8

UTF-32	UTF-16	UTF-8
00 07 BE	07 BE	DE BE
00 07 BF	07 BF	DE BF
00 07 C0	07 C0	DF 80
00 07 C1	07 C1	DF 81
.		
.		
.		
00 07 FE	07 FE	DF BE
00 07 FF	07 FF	DF BF
00 08 00	08 00	E0 A0 80
00 08 01	08 01	E0 A0 81
.		
.		
.		
00 08 3E	08 3E	E0 A0 BE
00 08 3F	08 3F	E0 A0 BF
00 08 40	08 40	E0 A1 80
00 08 41	08 41	E0 A1 81
.		
.		
.		
00 08 7E	08 7E	E0 A1 BE
00 08 7F	08 7F	E0 A1 BF
00 08 80	08 80	E0 A2 80
00 08 81	08 81	E0 A2 81
.		
.		
.		

Mappings, 9

UTF-32	UTF-16	UTF-8
00 08 BE	08 BE	E0 A2 BE
00 08 BF	08 BF	E0 A2 BF
00 08 C0	08 C0	E0 A3 80
00 08 C1	08 C1	E0 A3 81
.		
.		
.		
00 08 FE	08 FE	E0 A3 BE
00 08 FF	08 FF	E0 A3 BF
00 09 00	09 00	E0 A4 80
00 09 01	09 01	E0 A4 81
.		
.		
.		
00 09 3E	09 3E	E0 A4 BE
00 09 3F	09 3F	E0 A4 BF
00 09 40	09 40	E0 A5 80
00 09 41	09 41	E0 A5 81
.		
.		
.		

-- a big jump here, since the pattern is established --

00 0F FE	0F 7E	E0 BF BE
00 0F FF	0F 7F	E0 BF BF
00 10 00	10 00	E1 80 80
00 10 01	10 01	E1 80 81
.		
.		
.		

Mappings, 10

UTF-32	UTF-16	UTF-8
-- another big jump here, since the pattern is established --		
00 1B FE	1B 7E	E1 AF BE
00 1B FF	1B 7F	E1 AF BF
00 1C 00	1C 00	E1 B0 80
00 1C 01	1C 01	E1 B0 81
.		
.		
.		
-- another big jump here, since the pattern is established --		
00 1F FE	1F 7E	E1 BF BE
00 1F FF	1F 7F	E1 BF BF
00 20 00	20 00	E2 80 80
00 20 01	20 01	E2 80 81
.		
.		
.		
-- another big jump here, since the pattern is established --		
00 2F FE	2F 7E	E2 BF BE
00 2F FF	2F 7F	E2 BF BF
00 30 00	30 00	E3 80 80
00 30 01	30 01	E3 80 81
.		
.		
.		
-- another big jump here, since the pattern is established --		
00 3F FE	3F 7E	E3 BF BE
00 3F FF	3F 7F	E3 BF BF
00 40 00	40 00	E4 80 80
00 40 01	40 01	E4 80 81
.		
.		
.		

Mappings, 11

UTF-32 UTF-16 UTF-8

-- another big jump here, since the pattern is established --

```
00 4F FE           4F 7E        E4 BF BE
00 4F FF           4F 7F        E4 BF BF
00 50 00           50 00        E5 80 80
00 50 01           50 01        E1580 81
.
.
.
```

-- another big jump here, since the pattern is established --

```
00 D7 FE           D7 FE        ED 9F BE
00 D7 FF           D7 FF        ED 9F BF
```

-- at this point, we have reached the place where surrogate characters occur; a surrogate character by itself is not a valid Unicode character; we pick up again, after the surrogate points:

```
00 F9 00           F9 00        EF A4 80
00 F9 01           F9 01        EF A4 81
.
.
.
00 FF FE           FF FE        EF BF BE
00 FF FF           FF FE        EF BF BF
```

-- the next code point, x'010000', and all code points after this, will require pairs of surrogate characters for the UTF-16 vlaues ...

Mappings, 12

UTF-32	UTF-16	UTF-8
01 00 00	D8 00 DC 00	F0 90 80 80
01 00 01	D8 00 DC 01	F0 90 80 81
.		
.		
.		
01 01 FE	D8 00 DD FE	F0 90 87 BE
01 01 FF	D8 00 DD FF	F0 90 87 BF
01 02 00	D8 00 DE 00	F0 90 88 80
01 02 01	D8 00 DE 01	F0 90 88 81
.		
.		
.		
01 02 FE	D8 00 DE FE	F0 90 88 BE
01 02 FF	D8 00 DE FF	F0 90 88 BF

-- another big jump here, since the pattern is established --

01 E0 00	D8 01 DC 00	F0 90 90 80
01 E0 01	D8 01 DC 01	F0 90 90 81

-- another big jump here, since the pattern is established --

01 20 00	D8 40 DC 00	F0 90 90 80
01 20 01	D8 41 DC 01	F0 90 90 81

Mappings, 12

UTF-32	UTF-16	UTF-8
-- a final big jump here, up to the end of assigned characters		
0E 00 00	DB 40 DC 00	F3 A0 80 80
0E 00 01	DB 40 DC 01	F3 A0 80 81
.		
.		
.		
0E 01 EE	DB 40 DD EE	F3 A0 87 AE
0E 01 EF	DB 40 DD EF	F3 A0 87 AF