Appendix B: Implementation Guidelines

This section provides guidelines for implementing the Unicode character encoding scheme. It includes discussions of byte-ordering, special characters, compression, 7-bit transmission, sorting and searching, and conversion. The purpose of these guidelines is to promote good practice in Unicode implementations. As guidelines, they are *not* binding on the implementer and do not form part of the standard.

Byte Order Mark

Since Unicode plain text is a sequence of 16-bit codes, it is sensitive to the byte ordering which is used when writing text. Many processors place the least significant byte in the initial position, while others place the most significant byte in the initial position. Ideally, all Unicode would follow only one set of rules, but this would force one side to swap the byte order on reading and writing plain text files, even when the file never leaves the system on which it was created. To have an efficient way to indicate which byte order is used in a text, the Unicode standard has defined two values, U+FEFF BYTE ORDER MARK (BOM) and U+FFFE (not a character code), which are the byte-ordered mirror images of each other. The BYTE ORDER MARK is not a control character which selects the byte order of the text; rather its function is to insure recipients that they are looking at a correctly byte-ordered file. Furthermore, the sequence 0xFEFF is exceedingly rare at the outset of regular non-Unicode text files and may therefore easily serve as an implicit marker to identify a file as containing Unicode text. Strictly speaking, however, this constitutes a particular use of a Unicode character, and there is nothing in the standard itself that requires or endorses this usage. The BYTE ORDER MARK is completely ignored in all other text processing, including rendering, and can safely be removed from Unicode text without altering its interpretation. When it is used, the logical place for it is at the beginning of the text.

Special Character and Non-Character Values

U+FFFF and U+FFFE. These code points are *not* considered to be Unicode characters. They are therefore outside of any text which purports to be in Unicode encoding only. U+FFFF is reserved for private program use as a sentinel or other code. (U+FFFF is a short -1 in twos-complement notation.) Programs receiving this code, are not required to interpret it in any way. It is good practice, however, to recognize this code as an illegal condition and take appropriate error response action. U+FFFE is similar in all respects to U+FFFF, except that it is also the mirror image of U+FEFF BYTE ORDER MARK. (U+FFFE is a short -2 in twos-complement notation.) Therefore, its presence constitutes a strong hint that the text in question is byte-reversed. An appropriate error response could include trying to re-read the text after byte swapping it. Good practice would be to

at least let the user know that the text contained this code with the information that the text may be byte swapped.

ASCII Control Characters. The first thirty-two 16-bit characters in the Unicode standard are intended for encoding of the thirty-two ASCII control characters. Programs that conform to the Unicode standard can treat these control codes in exactly the same way as they treat their equivalents in ASCII. When converting control codes from existing 8-bit text, they are merely zero extended.

Escape Characters. In converting text with escape sequences to the Unicode character encoding, text must be converted to its Unicode equivalent. Converting escape sequences into Unicode on a character basis, (for instance, ESC-A turns into U+001B ESCAPE, U+0041 LATIN CAPITAL LETTER A), will allow the reverse conversion to be performed without forcing the program to recognize the escape sequence as such. In general, escape sequence should be translated into the Unicode encoding character by character.

Line and Paragraph Separator. The Unicode standard has two special characters U+2028 LINE SEPARATOR and U+2029 PARAGRAPH SEPARATOR. A new line is begun after each LINE SEPARATOR. A new paragraph is begun after each PARAGRAPH SEPARATOR. Since these are separator codes, it is not necessary either to start the first line or paragraph or to end the last line or paragraph with them. Doing so would indicate that there was an empty paragraph or line following. The paragraph separator can be inserted between paragraphs of text. Its use allows plain text files to be created which can be laid out on a different line width at the receiving end. The line separator can be used to indicate unconditional end of line. These are considered the canonical form of Unicode plain text.

Interaction with CR/LF. The Unicode standard does not prescribe specific semantics for U+000D CARRIAGE RETURN and U+000A LINE FEED. These codes are provided to represent any CR or LF characters employed by a higher level protocol, or retained in text translated from other standards. It is left to each application to interpret these codes as well as to decide whether to require their use, and whether CR/LF pairs or single codes are needed.

The Unicode Encoding as ANSI C wchar_t

With the wchar_t wide character type, ANSI C provides for inclusion of fixed width, wide characters. ANSI leaves the semantics of the wide character set to the specific implementation, but requires that the characters from the C execution set correspond to their wide character equivalents by zero extension. The Unicode characters equivalent to ASCII are the values from U+0020 to U+007F, which satisfy these conditions for 16-bit implementations of wchar_t.

Compression

Under some circumstances, using Unicode character encoding will double the amount of storage or memory space dedicated to the text portion of files. Compressing Unicode files may therefore be

an attractive option. There are commercially available compression algorithms, such as LZW, that will compress files to something near their theoretical minimum. That means, if a particular text in ASCII took a certain number of bytes to encode, the same text expressed in a Unicode encoding can be compressed to the size as if the compression had been applied to the ASCII file directly. Since Unicode text is composed from a 16-bit token set, algorithms such as LZW, which are sensitive to the width of the individual tokens, may stand to gain from being reimplemented based on 16-bit tokens.

The Unicode codespace is arranged such that characters within the same script are contiguous as much as possible. An efficient compression algorithm might run-length encode the most significant byte. However, compression employed at a level where it is visible to the text processing parts of the program reintroduces the kind of complexities found in multibyte or or other stateful encodings, which the Unicode character encoding was designed to avoid. Compression by definition does not conform to the Unicode standard; rather it constitutes a higher-level protocol.

Where compression is built into the underlying support layer, such as modem transmission protocols, it can be effective in eliminating the size overhead of a Unicode encoding without the cost of added complexity.

7-bit Transmission

Some transmission protocols use ASCII control codes for flow control. Others, including some UNIX mailers are notorious for their restrictions to 7-bit ASCII. In these cases, Unicode transmissions must be encapsulated. A number of encapsulation protocols exist today, such as UUENCODE and BTOA. These can be combined with compression in the same pass, reducing the transmission overhead.

Converting To and From Canonical Form

The Unicode standard contains explicit codes for the most frequently used accented characters. These characters can be composed from elements; in the case of accented letters, these are the base characters and non-spacing marks. The Unicode standard provides a table of suggested spellings (decompositions) of characters that can be composed of a base character plus a non-spacing mark. These tables can be used to unify spelling. As far as the Unicode standard is concerned, both canonical and composed spellings are equivalent, and there is certainly no need to retain a mixed mode spelling.

If the purpose of a program is essentially one of pass-through, it may not be advisable to use composed characters, especially since less ambitious programs may not be able to interpret non-spacing marks. However, the greatest generality may be obtained through use of composition, or by allowing both forms.

Characters Not Used in a Subset

The Unicode standard does not require that an application be capable of interpreting and rendering *all* of the Unicode characters in order to be conformant. Many systems will have fonts only for some scripts, but not for others; sorting and other text processing rules may be implemented only for a limited set of languages. There is therefore a subset of characters which an implementation is able to interpret.

In general, it is nonconformant to modify or remove any Unicode characters that are outside of the program's subset of implemented Unicode characters. This is especially true for data which is intended to be transmitted through to other applications.

The Unicode standard explicitly disallows coding such as the following:

```
char ch = getchar() & 0x7f;
```

Conformance to the Unicode standard *requires* that whenever characters are to be re-transmitted, that unreadable characters must not be removed. For a more detailed discussion of this issue, see Section 2.6 on Conformance.

Sorting

Only those aspects of sorting are discussed here that relate to the question of character encoding. Much of the information in this section is also applicable to searching, especially when the goal is not an exact match, but a case-insensitive or other near match.

Culturally-Expected Sorting

There is usually not one, but several possible types of sort order possibilities that vary from culture to culture. Rarely is there a one-to-one correspondence of character codes to sort methods.

As a result, it is neither possible to arrange characters in an encoding in the correct order, nor is it possible to provide single level sort weight tables. Therefore, character encoding details have only an indirect impact on culturally-expected sorting.

Sort order can be by word or sentence, case sensitive or insensitive, ignoring accents or not; it can also be phonetic or based on the identity of the character, such as ordering by stroke and radical for East Asian ideographs. Phonetic sorting of Han characters requires use of a look-up dictionary of words, or special programs to maintain a separate phonetic spelling for the words in the text.

Languages vary not only on which types of sorts to use (and in which order they are to be applied), but also in what constitutes a fundamental element for sorting. Swedish treats LATIN LETTER A DIAERESIS as an individual letter sorting after z in the alphabet, whereas German sorts it either like "æ" or like other accented forms of "ä" following a. Spanish sorts the digraph "ch" as if it were a letter between "c" and "d." Examples from other languages (and scripts) abound.

To address the complexities of culturally-expected sorting, a multi-pass sorting algorithm is typically employed.[1] In its first pass, several categories of weights are accumulated for each character in the sort string. Categories include alphabetic, case and diacritic weights, among others. At the end of the first pass, the sort key contains a string of alphabetic weights, followed by a string of case weights and so on. These substrings are then compared one by one, so that case and accent differences can be ignored, or applied only where needed to differentiate otherwise indentical sort strings. The first pass of this scheme looks very similar to the Unicode decomposition into base-form and accent. The fact that the Unicode standard allows multiple spellings (composed and composite) of the same accented letter, turns out not to matter at all. If anything, a completely decomposed text stream might simplify the first implementation of sorting.

Handling Non-spacing Marks

A fixed set of composed character sequences can be rendered effectively by means of fairly simple substitution. Wherever a sequence (of base character, non-spacing mark) occurs, a glyph representing the combined form can be substituted. In simple, fixed-width character rendering, a non-spacing mark has a zero advance width, and a composed character sequence will automatically have the same width as the base character. When truncating strings, it is easiest always to truncate starting from the end and working backwards. A trailing non-spacing mark will then not be separated from the preceeding base character.

More sophisticated text rendering systems may take further measures to account for those cases where the composed character sequence has a different advance width than the base character. Such systems can also supply more sophisticated truncation routines.

When rendering a sequence of more than one non-spacing mark, the non-spacing marks should be stacked outwards from the base character. That is, if two non-spacing marks appear over a base character, then the first non-spacing mark should appear on top of the base character, and the second non-spacing mark on top of the first. If two non-spacing marks appear under a base character, then the first non-spacing mark should appear beneath the base character, and the second non-spacing mark below the first. (See Section 2.5, "General Principles Governing Non-Spacing Marks.")

If there is an unknown composed character sequence which is outside of a fixed, renderable set, then there are several methods of dealing with it. One method indicates the inability to draw the sequence by first drawing the base character, and then rendering the non-spacing mark as floating on a dotted circle, as illustrated in the code charts. Another method is to use a default fixed positioning of the non-spacing mark, generally placed away from overlap with possible base characters. For example, the default positioning of a circumflex can be above the ascent, which will place it

1. A good example can be found in Denis Garneau, *Keys to Sort and Search for Culturally-Expected Results* (IBM document number GG24-3516, June 1, 1990), which addresses the problem for western European languages, Arabic and Hebrew.

above capitals. Even though this will not yield a particularly attractive result for letters such as *g-circumflex*, it should generally be recognizable. More sophisticated systems can provide better rendering for composed character sequences.

Correct multilingual comparison routines must already be able to compare a sequence of characters as one, or one character as if it were a sequence. Such routines can also handle composed character sequences when supplied the appropriate data. When searching strings, remember to check for additional non-spacing marks in the target string that may affect the interpretation of the last matching character. Line-break algorithms generally use state machines for determining word breaks. Such algorithms can also be adapted to prevent separation of non-spacing marks from base characters.