1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Transition from ANSI to Unicode: Multilingual Support in Operating Systems and Programming Languages

Pei-Chi Wu

Department of Information Management
Department of Computer Science and Information Engineering
National Penghu Institute of Marine & Management Technology
300 Liu-Ho Road, Makung City, Penghu, Taiwan 880, R.O.C.
Email: pcwu@npit.edu.tw

ABSTRACT

Character sets are one of basic issues for information interchange. Most current character sets extend ANSI's 7-bit character set. These extensions are conflicted with each other and make the design of multilingual information systems complicated. Unicode or Universal Character Set (UCS) is a character set that covers symbols in major written languages. Text files and strings usually have no header to indicate which character set is in use, and they currently use ANSI by default. The transition from ANSI to Unicode may last a longer time than expected. This paper presents the following methods to help the transition: 1) A text file format of fixed-width characters 2) A tagged string storage: Each string has a tag representing which character set or coding format is in use. 3) A method for assigning the format of string literals. These methods can improve multilingual support without introducing much complexity.

Keywords: character sets, text files, control codes, byte order, string literals, source files.

1. INTRODUCTION

Character sets are one of basic issues for information interchange. Most current character sets extend ANSI's 7-bit character set (ASCII) [1]. For example, countries of East Asia use double-byte character sets: ANSI characters $(00)_{16}$–$(7F)_{16}$ are represented in single-byte, but some code points of $(80)_{16}$–$(FF)_{16}$ are used as leading bytes. A double-byte character is represented by a leading byte in addition with a trailing byte. Such ANSI extensions are national standards of many countries. These extensions are also informally called "ANSI" character sets by some manufacturers; these actually are different extensions of ANSI. These extensions are conflicted with each other and make the design of multilingual information systems complicated. One of the major problems is mixing single-byte and double-byte characters in one character stream. Each double-byte ANSI extension uses a different set of code points for leading bytes, so information systems should be tailored to each ANSI extension.

Example multilingual information systems include library automation systems and global information environments, such as Internet. Professionals of information processing are thus seeking for a character set that can be shared by all countries. Unicode [8] is such a character set that covers symbols in major written languages. Unicode is a fixed-width coding architecture: a 16-bit code point represents one character. Unicode unifies conflicted ANSI extensions into one coding architecture. Many manufacturers now support Unicode, or at least claim to adopt Unicode. It is expected that Unicode will be widely accepted for use in information interchange. Unicode is a subset of ISO/IEC 10646 Universal Character Set (UCS), a 31-bit coding architecture. Unicode is UCS's 0-plane: Basic Multilingual Plane. ISO/IEC 10646 defines two alternative forms: UCS-4 (4-byte) and UCS-2 (2-byte, i.e., Unicode). Although ISO/IEC has not yet filled planes other than 0, information systems should better prepare for such movements.

Due to the fast rising of Internet applications, multilingual support in operating systems becomes more and more important than ever before. Transiting to a new character set influences both storage formats of text data and programs of text processing. Although manufacturers of information systems start to support Unicode and UCS, contents encoded with Unicode are still in rare use. Text files and strings usually have no header to indicate which character set is in use. Text files currently use ANSI character set by default. The transition from ANSI to Unicode may last a longer time than expected.

This paper presents the following methods to help the transition from ANSI to Unicode: 1) A text file format of fixed-width characters: If the first character in a text file is a nonzero control code, such as U+007F (DEL), the file is in UCS; otherwise, it is in ANSI. This control code is called *UCS control code*. The control code indicates which UCS format or byte order is in use. 2) A tagged string storage: Each string has a tag representing which character set or coding format is in use, e.g., ANSI, 8-bit subset of UCS-2, UCS-2, or UCS-4. 3) A method for assigning the format of string literals in source files: String literals of various character sets use the same syntax notation, and their storage formats are the same as that of their source files. These methods can improve multilingual support in operating systems and programming languages without introducing much complexity.

2. Related Work

One of the problems in adopting Unicode is that Unicode doubles the storage space for characters in ASCII and ISO 8859-1 (ISO Latin-1), which are originally encoded in single-byte. Another problem is that Unicode mixes characters in several languages. For example, a Han character both in Chinese and Japanese is represented by one code point. This complicates information processing that is dedicated to one language only. Mudawwar [7] argues this issue and proposes Multicode, which is similar to switching between code pages of various languages.

Recent Internet protocols are designed with multilingual support. XML [9] provides a means for declaring encoding formats, for example:

<?xml encoding='UTF-8'?>

XML processors can automatically detect the encoding format by reading the first characters '<?xml' in the encoding declaration [9, Appendix F]. However, if a program changes the XML entity's encoding format without updating the encoding declaration, the storage and the declaration mismatch.

Character sets are also an important issue in programming languages. Strings in Java language [3] use Unicode. Current Java virtual machines [6] adopt UTF-8 encoding format [8, p. A-7] for Unicode characters. The C [5] and C++ [2] languages provide two data types to handle characters of different widths: conventional characters (char) and wide characters (wchar_t). There are also two kind of string literals:

"This is a character string literal" and

L"This is a wide-character string literal".

This complicates the design of programming languages and their related libraries. In string functions of the C standard library, there are versions for both character data types, for example:

```
double strtod( const char *nptr, char **endptr );
double wcstod( const wchar_t *nptr, wchar_t **endptr );
```

Both functions convert a string (char* or wchar_t*) to a double-precision floating-point numbers (double).

Visual C++ 2 run-time libraries contain single-byte, multi-byte, and Unicode versions of functions that take parameters of strings [4, Ch. 3]. The function prototypes resolve to single-byte functions by default. If the compile-time flag _UNICODE is defined, the prototypes resolve to wide-character functions. If the _MBCS flag is defined, the prototypes resolve to multi-byte functions. Visual C++ also defines _TCHAR as the generic type of characters.

UTF-8 has been served as an intermediate format for transiting from ANSI to Unicode. UTF-8 is an encoding of Unicode into 8-bit characters. It is variable-length: Each code value (non-surrogate) is represented in 1, 2, or 3 bytes. Although UTF-8 in single-byte form is the same as ASCII, it is incompatible with ISO 8859-1. Each Han character, which originally occupies 2 bytes when encoded in Big-5 or GB, is expanded to 3 bytes. This wastes storage space and complicates the processing of Han characters. UTF-8 is best suitable only for English text data, which are already best served by ASCII. In addition, UTF-8 cannot support encoding of UCS-4. Thus, it is questionable that UTF-8 will be widely adopted in multilingual information systems.

3. Text File Format

Text files usually have no header to indicate which character set or encoding format is in use. A text file contains just a series of characters. Most text files currently use ANSI character set by default. It is almost impossible for so many programs that use text files to agree on one text file format, such as adding a declaration of encoding formats like XML. In addition, a text file format should provide a space-efficient means for characters in ASCII, ISO 8859-1, CJK, etc. The header in the text file format should be kept as minimal. This file format should also be applicable to UCS-4.

Our text file format is outlined below:

1. Providing an 8-bit fixed-width format: The 8-bit format uses the subset of UCS-2 U+0000–U+00FF. There is no way to switch between 8-bit and 16-bit formats. The 8-bit subset includes Basic Latin (U+0000–U+007F) and Latin-1 Supplement (U+0080–U+00FF). The former is ASCII; the latter is ISO 8859-1. This subset covers languages of most western countries. In the following, to have a name like UCS-2 and UCS-4, this subset is also called UCS-1.
2. Using a control code to distinguish ANSI and UCS formats: If the first character in a text file is a nonzero control code, such as U+007F (DEL), the file is in UCS; otherwise, it is in ANSI. This special control code is called *UCS control code*. When the UCS control code is in single-byte, it denotes that the text file is in 8-bit subset of UCS-2. Double-byte denotes UCS-2; quad-byte denotes UCS-4. The byte order of UCS data has the same order as UCS control code.

Use UCS-2 and control code U+007F as an example: If the first 2 bytes are $(00)_{16}$ $(7F)_{16}$, they denote Big-Endian (high-byte first). The first 2 bytes $(7F)_{16}$ $(00)_{16}$ denote Little-Endian (low-byte first).

This method is simple yet powerful:

1. 8-bit UCS format: 8-bit character sets such as ISO 8859-1 have been used in many countries; however, they are conflicted with double-byte character sets in East Asia. Text files in ISO 8859-1 can be inserted with the 8-bit UCS control code, and then become 8-bit UCS text files. This storage format can solve the space expansion problem, which occurs when files in ASCII and ISO 8859-1 are converted to Unicode. This method is also superior to UTF-8 format, which 8-bit format covers only ASCII.
2. Fixed-width encoding: Although there are three encoding widths for UCS, these encodings do not mix themselves. Fixed-width encoding is the simplest and most efficient format for character processing.
3. Distinguishing storage formats: The UCS control code can distinguish ANSI and three UCS formats. Text files having the UCS control code are in UCS; otherwise, they are in ANSI. For example, use U+007F (DEL) as the UCS control code. Since $(00)_{16}$ and $(7F)_{16}$ do not appear in typical ANSI text files, they can be used for UCS applications to distinguish UCS and ANSI data. This can avoid loading text data erroneously. When the UCS control code is represented in *k*-byte, it denotes *k*-byte storage format in a text file.
4. Byte ordering: Unicode allows two kind of byte orders. The standard suggests using U+FEFF (byte order mark) and U+FFFE (non-character) to detect the byte order and to indicate that the text file contains Unicode text. However, $(FE)_{16}$ and $(FF)_{16}$ have already been used in other character sets. For example, ISO 8859-1 defines these codes as 'þ' and 'ÿ'. Although it is unlikely that a file begins with these characters, using an ASCII control code may be better.
5. Aligned addresses: Most modern computers allow only addresses aligned with data widths: When a program accesses *k*-bit data, the address must divide *k*. In our method, UCS control code has the same width as that of other characters. Thus, if the buffer of a text file starts at an address dividing 4, there is no problem to load and access characters of 1, 2, or 4 bytes.
6. Minimal changes: UCS control code is also a character, so our method does not change the definition of text files: files containing a series of characters.

Since UCS control code is important, it is necessary to allocate an ANSI control code dedicated for this purpose. UCS control code should meet the following criteria: (a) 8-bit control code, represented in one byte and not conflicted with typical ANSI data; (b) non-zero, to denote byte orders. The control code $(7F)_{16}$ meets both criteria. Control codes $(00)_{16}$ (NULL) and $(7F)_{16}$ (DEL) currently have no real effect in text files. NULL originally represents the unused space in a punched paper tape; DEL represents the mark of all holes for deleting a punched character. Most other ASCII control codes have real effects in text files. Control codes $(81)_{16}$–$(9F)_{16}$ have been used as leading bytes of double-byte character sets in code pages 932 (Japanese) and 949 (Korean). Only $(80)_{16}$ is available. However, there may exist sloppy programs that test only the most significant bit when detecting leading bytes. These programs will erroneously treat $(80)_{16}$ as a leading byte.

In according to byte orders and three UCS formats, totally there are 6 cases in the header of a text file: (A denotes ANSI data; U denotes UCS data; each letter or digit denotes 4 bits; 7F and 00 are control codes)

AA ...: ANSI data, containing no UCS control code.
7F UU ...: 8-bit USC-2 subset.
00 7F UU UU ...: UCS-2, Big Endian.
7F 00 UU UU ...: UCS-2, Little Endian.
00 00 00 7F UU UU UU UU ...: UCS-4, Big Endian.
7F 00 00 00 UU UU UU UU ...: UCS-4, Little Endian.

Our method does not mix different encoding formats in a text file. Sometimes this may waste space. For example, consider a document in English with a Chinese abstract. The document will be encoded in UCS-2, and all ASCII characters in the document are encoded in double-byte. To reduce space, the Chinese abstract can be stored separately in another file. On the other hand, typical Chinese text files may contain characters in ASCII. However, in these files, the number of such characters is usually relatively small.

4. Strings Tagged with Character Set

There are two kind of strings in programming languages: null-terminated strings (e.g., in C and C++) and strings tagged with lengths (e.g., in Basic and Java). In these strings, the character set or encoding format in use is implicitly defined. There is no way to specify a code page number or an encoding format ID in a string. These storage formats work fine when there is only one standard character set and encoding format, but they cannot handle more than one character set or encoding format.

We propose a tagged string storage: Each string has a tag indicating which character set or encoding format is in use, e.g., ANSI, UCS-1, UCS-2, or UCS-4. Figure 1 shows class String in Java-like syntax. Class String has three data fields: tag, length, and contents. The tag can be 0 (ANSI), 1 (UCS-1), 2 (UCS-2), and 3 (UCS-4). The length is the number of characters. The contents are a byte array, whose size is determined by the tag and the length of the string. For UCS subsets and single-byte ANSI extensions, size = *nbytes*(tag) · length, where *nbytes*(0..3) are the widths (in bytes) of character sets: 1, 1, 2, and 4, respectively. For double-byte ANSI extensions, length ≤ size ≤ 2 · length. The ansi, ucs1, ucs2, ucs4 are conversion functions for corresponding encoding formats. When a conversion fails, class String raises ConversionFailed exception. Operator '[]' accesses a character in a string by an index and returns a Code, which is implemented as an unsigned integer.

```
public class String extends Object  {
  private int tag;
  private int length;
  private byte contents[size];
  public Code  operator[](int index);
  public String operator+
      (String a,  String b);
  public String ansi()
      throws ConversionFailed;
  public String ucs1()
      throws ConversionFailed;
  public String ucs2()
      throws ConversionFailed;
  public String ucs4()
      throws ConversionFailed;   ...
}
```

Figure 1. The interfaces of class String.

Operator '+' can concatenate strings of any tag. To avoid conflicts in ANSI and UCS, we prefer using a UCS format as the resulting format when concatenating ANSI and UCS strings. The following shows the concatenation of strings in various tags:

ANSI + ANSI → ANSI;
UCS-2 + UCS-2 → UCS-2;
ANSI + UCS-2 → UCS-2;
UCS-1 + UCS-2 → UCS-2;
UCS-2 + UCS-4 → UCS-4.

The tag of the resulting string can be different in according to which ANSI extension is in use:

ANSI + UCS-1 → UCS-1; or
ANSI + UCS-1 → UCS-2.

The former is for single-byte ANSI extensions; the latter is for double-byte ANSI extensions.

In C/C++ languages, the length in class String can be omitted, but the end of the contents must be appended with a null character. To obtain a string storage format suitable for various programming languages, we combine both structures as shown in Figure 2. |NULL| denotes the size of a null character.

```
Public class String extends Object  {
  int tag;
  int length;
  byte contents[size + |NULL|];  ...
}
```

Figure 2. A string storage appended with a null character.

If the value of a tags is restricted to ANSI, UCS-1, UCS-2, and UCS-4, a tag occupies only 2 bits. We can allocate the tag and the length both in bit fields or in bits of a 32-bit integer, called tag_length. Let the tag be in the high bits and the length be in the low bits. When tag = 0 (ANSI), this string format is compatible to conventional ANSI strings, just having a slightly limited string length (2 bits less).

```
Public class String extends Object  {
       int tag:2;
       int length:30;  ...
}
or:
public class String extends Object  {
       int tag_length;     ...
}
```

Figure 3. The interfaces of string with bit fields.

The design of library routines can also be simplified. For example:

int to_int(String s);

double to_double(String s);   ...

String functions such as to_double() can be applied to strings in any character set or encoding format. When the value of the tag extends to other character sets or encoding formats, e.g., UTF-7, UTF-8, and UTF-16 [8, Appendix A], the interface of these functions remains unchanged.

There are many standard system functions that take parameters of strings. Consider the following C functions:

FILE* fopen(const char *filename, const char *mode);

int rename(const char *oldname, const char *newname);

The data type char* can only represent strings of one specific character set or encoding format. This limitation can be removed by replacing char* with tagged strings:

FILE* fopen(const String filename, const String mode);

int rename(const String oldname, const String newname);

Operating systems for western countries can use UCS-1; systems for countries in East Asia can use UCS-2. Both versions of operating systems provide the same interface for these system functions, although their internal implementations may be different. This method simplifies application program interfaces. Operating systems can provide just one set of application program interface instead of two: one for ANSI and another for Unicode.

5. String Literals in Source Files

Programming languages that provide more than one kind of strings or character types should provide a means for assigning the format of string literals. In C and C++ languages, string literals of wide characters are denoted with 'L'. Unfortunately such notation does not work very well. In this section, we propose a simple means for automatically assigning the format of string literals in source files.

In C and C++ languages, let conventional characters be in ANSI, wide characters be in UCS-2, and the source files (usually text files) be in ANSI. Consider the following cases:

1. "This is a character string literal"
2. L"This is a wide-character string literal"
3. "這是一般字元的字串常數"
4. L"這是一寬字元的字串常數"

Cases 1 and 3 are conventional characters (char), so the compiler takes no conversion when reading these string literals from source files and writing them to object files. In case 2, the compiler needs to convert single-byte ANSI data to UCS-2: appending each character with a 0 at the high byte. In case 4, the compiler needs to convert double-byte ANSI data (e.g., traditional Chinese in Big-5 character set) to UCS-2. The case 4 is the most difficult to handle without operating systems support, such as standard conversion functions for various character sets and encoding formats.

One way to solve this problem is completely eliminating multilingual string literals from source files. The following coding practice has been strongly suggested for writing global software [4, Ch.2]: Any element of a program that requires translation for different languages should be separated from source programs. Following this coding convention, most string literals left in source files will be in 7-bit ASCII. However, the problem is just left to so called resource editors, which handle strings and graphical user interface components. Resource editors are usually platform-dependent, while major programming languages have international standards. Thus, it is still better to be able to solve the problem inside programming languages rather than left to programming environments.

Instead of using syntax notation to distinguish string literals of various formats, we use the character set and the encoding format of source files. All string literals use the same syntax notation, and their storage format is the same as that of their source files. In a source file having the first character to be a UCS control code, all its string literals are in the format of UCS; otherwise, they are in ANSI. Consider the following examples:

File 1: ... "This is a character string literal" ...

File 2: 0x7F 0x0 ... "This is a wide-character string literal"

File 3: ... "這是一般字元的字串常數"

File 4: 0x7F 0x0 0x0 0x0 ... "這是一寬字元的字串常數" ...

In File1 and 3, their first characters are not the UCS control code, so they are ANSI files and all string literals are in ANSI. File 2 begins with the double-byte UCS control code (0x007F), so it is a UCS-2 file and all string literals are in UCS-2. File 4 begins with the quad-byte UCS control code (0x0000007F), so it is a UCS-4 file and all string literals are in UCS-4.

Using the format of source files to determine the format of string literals makes the programming languages and their compilers independent of character sets and encoding formats. String literals have the same encoding format when they are in source files and when they are used at run-time. Compilers only need to handle the byte order of string literals, when the byte order used in source files is different from the byte order of the target machine.

Converting a program to a new character set is also easy: just convert the character set or the encoding format of the source files. For example, consider converting all string literals in a program from ANSI to Unicode. There is no need to apply any special programming tool to denote each string literal with an 'L'. Any ANSI-to-Unicode conversion utility will work much better for this purpose.

6. Conclusions and Future Work

This paper has presented methods to help the transition from ANSI to Unicode. Firstly, we have presented a text file format, which provides three UCS subsets, fixed-width encoding, byte orders, and aligned addresses. Secondly, we have presented a string tagged with the character set or encoding format in use. Using this string data type, operating systems and standard libraries of programming languages can provide just one set of program interfaces for functions taking parameters of strings. Thirdly, we have presented a method that assigns the format of string literals to be the same as that of their source files. This makes programming languages and their compilers independent of character sets and encoding formats. Converting a program to a new character set is also easy: just convert the character set or the encoding format of the source files.

The methods presented here can influence the design of operating systems, programming languages, and compilers. Multilingual support in operating systems and programming languages can be achieved without introducing much complexity to these system programs, which are already very complicated today. Transiting from ANSI to Unicode can be smoothly. Multilingual applications built on this technique can work with ANSI data and be ready for future adoption of UCS-2 and UCS-4.

References

[1] American National Standards Institute, *Coded character set - 7-bit American national standard code for information interchange*, New York, 1986 (ANSI X3.4 - 1986).

[2] Ellis, M.A., and Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley, Massachusetts, 1990.

[3] Flanagan, D., *Java in a Nutshell*, O-Reilly & Associates, Inc., 1996.

[4] Kano, N., *Developing International Software for Windows 95 and Windows NT*, Microsoft Press, 1995.

[5] Kernighan, B.W., and Ritchie, D. M., *The C Programming Language*, 2nd Ed., Prentice-Hall, New Jersey, 1988.

[6] Meyer, J., and Downing, T., *Java Virtual Machine*, O'Reilly & Associates, Inc., 1997.

[7] Mudawwar, M. F., "Multicode: A Truly Multilingual Approach to Text Encoding," *IEEE Computer*, Vol. 30, No. 4, April 1997, pp. 37-43.

[8] The Unicode Consortium, *The Unicode Standard*, Version 2.0, Addison-Wesley, Reading, Massachusetts, 1996.

[9] World-Wide Web Consortium, *Extensible Markup Language (XML)*, Version 1.0, W3C Recommendation, Feb. 10, 1998.