UNICODE: A UNIVERSAL CHARACTER CODE

By Jürgen Bettels and F. Avery Bishop

ABSTRACT

A universal character encoding is required to produce software that can be localized for any language or that can process and communicate data in any language. The Unicode standard is the product of a joint effort of information technology companies and individual experts; its encoding has been accepted by ISO as the international standard ISO/IEC 10646. Unicode defines 16-bit codes for the characters of most scripts used in the world's languages. Encoding for some missing scripts will be added over time. The Unicode standard defines a set of rules that help implementors build text-processing and rendering engines. For Digital, Unicode represents a strategic direction in internationalization technology. Many software-producing companies have also announced future support for Unicode.

INTRODUCTION

A universal character encoding -- the Unicode standard -- has been developed to produce international software and to process and render data in most of the world's languages. In this paper, we present the background of the development of this standard among vendors and by the International Organization for Standardization (ISO). We describe the character encoding's design goals and principles. We also discuss the issues an application handles when processing Unicode text. We conclude with a description of some approaches that can be taken to support Unicode and a discussion of Microsoft's implementation. Microsoft's decision to use Unicode as the native text encoding in its Windows NT (New Technology) operating system is of particular significance for the success of Unicode.

BACKGROUND

In the 1980s, software markets grew throughout the world, and the need for a means to represent text in many languages became apparent. The complexity of writing software to represent text hindered the development of global software.

The obstacles to writing international software were the following.

1. Stateful encoding. The character represented by a particular value in a text stream depended on values earlier in the stream, for example, the escape sequences of the ISO/IEC 2022 standard.[1]

2. Variable-length encoding. The character width varied from one to four bytes, making it impossible to know how many

characters were in a string of a known number of bytes, without first parsing the string.

3. Overloaded character codes and font systems. Character codes tended to encode glyph variants such as ligatures; font architectures often included characters to enable display of characters from various languages simply by varying the font.

In the 1980s, character code experts from around the world began work on two initially parallel projects to eliminate these obstacles. In 1984, the ISO started work on a universal character encoding. This effort placed heavy emphasis on compatibility with existing standards. The ISO/IEC committee published a Draft International Standard (DIS) in spring 1991.[2] By that time, the work on Unicode (described in the next section) was also nearing completion, and many experts were alarmed by the potential for confusion from two competing standards. Several of the ISO national bodies therefore opposed adoption of the DIS and asked that ISO and Unicode work together to design a universal character code standard.

The Origins of Unicode

In some sense Unicode is an offshoot of the ISO/IEC 10646 work. Peter Fenwick, one of the early conveners of the ISO working group responsible for 10646, developed a proposal called "Alternative B," based on a 16-bit code with no restriction on the use of control octets. He presented his ideas to Joseph Becker of Xerox, who had also been working in this area.[3]

In early 1988, Becker met with other experts in linguistics and international software design from Apple Computer (notably Lee Collins and Mark Davis) to design a new character encoding. As one of the original designers, Becker gave this code the name Unicode, to signify the three important elements of its design philosophy:

1. Universal. The code was to cover all major modern written languages.

2. Unique. Each character was to have exactly one encoding.

3. Uniform. Each character was to be represented by a fixed width in bits.

The Unicode design effort was eventually joined by other vendors, and in 1991 it was incorporated as a nonprofit consortium to design, promote, and maintain the Unicode standard. Today member companies include Aldus, Apple Computer, Borland, Digital, Hewlett-Packard, International Business Machines, Lotus, Microsoft, NeXT, Novell, The Research Libraries Group, Sun Microsystems, Symantec, Taligent, Unisys, WordPerfect, and Xerox.

Version 1.0, volume 1 of the 16-bit Unicode standard was published in October 1991, followed by volume 2 in June 1992.[4,5]

It was sometimes necessary to sacrifice the three design principles outlined above to meet conflicting needs, such as compatibility with existing character code standards. Nevertheless, the Unicode designers have made much progress toward solving the problems faced in the past decade by designers of international software.

The Merger of 10646 and Unicode

Urged by public pressure from user groups such as IBM's SHARE, as well as by industry representatives from Digital, Hewlett-Packard, IBM, and Xerox, the ISO 10646 and Unicode design groups met in August 1991; together they began to create a single universal character encoding. Both groups compromised to create a draft standard that is often referred to as Unicode/10646. This draft standard was accepted as an international character code standard by the votes of the ISO/IEC national bodies in the spring of 1992.[6]

As a result of the merger with ISO 10646, the Unicode standard now includes an errata insert called Unicode 1.0.1 in both volumes of version 1.0 to reflect the changes to character codes in Unicode 1.0.[7] The Unicode Consortium has also committed to publish a technical report called Unicode 1.1 that will align the Unicode standard completely with the ISO/IEC 10646 two-octet compaction form (the 16-bit form) also called UCS-2.

Relationship between Unicode and ISO/IEC 10646

Unicode is a 16-bit code, and ISO/IEC 10646 defines a two-octet (UCS-2) and a four-octet (UCS-4) encoding form. The repertoire and code values of UCS-2, also called the base multilingual plane (BMP), are identical to Unicode 1.1. No characters are currently encoded beyond the BMP; the UCS-4 codes defined are the two UCS-2 octets padded with two zero octets. Although UCS-2 and Unicode are very close in definition, certain differences remain.

By its scope, ISO/IEC 10646 is limited to the coding aspects of the standards. Unicode includes additional specifications that help aspects of implementation. Unicode defines the semantics of characters more explicitly than 10646 does. For example, it defines the default display order of a stream of bidirectional text. (Hebrew text with numbers or embedded text in Latin script is described in the section Display of Bidirectional Strings.) Unicode also provides tables of character attributes and conversion to other character sets.

In contrast with the Unicode standard, ISO 10646 defines the following three compliance levels of support of combining

characters:

- Level 1. Combining characters are not allowed (recognized) by the software.
- Level 2. This level is intended to avoid duplicate coded representations of text for some scripts, e.g., Latin, Greek, and Hiragana.
- Level 3. All combining characters are allowed.

Therefore, Unicode 1.1 can be considered a superset of UCS-2, level 3.

Throughout the remainder of this paper, we refer to this jointly developed standard as Unicode. Where differences exist between ISO 10646 and Unicode standards, we describe the Unicode functionality. We also point out the fact that Unicode and ISO sometimes use different terms to denote the same concept. When identifying characters, we use the hexadecimal code identification and the ISO character names.

GENERAL DESIGN OF UNICODE

This section discusses the design goals of Unicode and its adherence to or variance from the principles of universality, uniqueness, and uniformity.

Design Goals and Principles

The fundamental design goal of Unicode is to create a unique encoding for the characters of all scripts used by living languages. In addition, the intention is to encode scripts of historic languages and symbols or other characters whose use justifies encoding.

An important design principle is to encode each character with equal width, i.e., with the same number of bits. The Unicode designers deliberately resisted any calls for variable-length or stateful encodings. Preserving the simplicity and uniformity of the encoding was considered more important than considerations of optimization for storage requirements.

A Unicode character is therefore a 16-bit entity, and the complete code space of over 65,000 code positions is available to encode characters. A text encoded in Unicode consists of a stream of 16-bit Unicode characters without any other embedded controls. Such a text is sometimes referred to as Unicode plain text. The section Processing Unicode Text discusses these concepts in more detail.

Another departure from the traditional design of code sets is

Unicode's inclusion of combining characters, i.e., characters that are rendered above, below, or otherwise in close association with the preceding character in the text stream. Examples are the accents used in the Latin scripts, as well as the vowel marks of the Arabic script. Combining characters are allowed to combine with any other character, so it is possible to create new text elements out of such combinations.[8] This technique can be used in bibliographic applications, or by linguists to create a script for a language that does not yet have a written representation, or to transliterate one language using the script of another. An example in recent times is the conversion of some Central Asian writing systems from the Arabic to the Latin script, following Turkey's example in the 1920s (Kazakhstan).

An additional design principle is to avoid duplication of characters. Any character that is nearly identical in shape across languages and is used in an equivalent way in these languages is assigned a single code position. This principle led to the unification of the ideographs used in the Chinese, Japanese, and Korean written languages. This so-called CJK unification was achieved with the cooperation of official representatives from the countries involved.

The principle of uniqueness was also applied to decide that certain characters should not be encoded separately. In general, the principle states that Unicode encodes characters and not glyphs or glyph variations. A character in Unicode represents an abstract concept rather than the manifestation as a particular form or glyph. As shown in Figure 1, the glyphs of many fonts that render the Latin character A all correspond to the same abstract character "a."

[Figure 1 (Abstract Latin Letter "a" and Style Variants) is not available in ASCII format.]

Another example is the Arabic presentation form. An Arabic character may be written in up to four different shapes. Figure 2 shows an Arabic character written in its isolated form, and at the beginning, in the middle, and at the end of a word. According to the design principle of encoding abstract characters, these presentation variants are all represented by one Unicode character.[9]

[Figure 2 (Isolated, Final, Initial, and Middle Forms of the Arabic Character Sheen) is not available in ASCII format.]

Since much existing text data is encoded using historic character set standards, a means was provided to ensure the integrity of characters upon conversion to Unicode. Great care was taken to create a Unicode character corresponding to each character in existing standards. Characters identical in shape appearing in different standards are identified and mapped to a single Unicode character. For characters appearing twice in the same standard, a compatibility zone was created. These characters are encoded as

required to make round-trip conversion possible between other standards and Unicode. The Unicode Consortium has agreed to create mapping tables for this purpose.

Text Elements and Combining Characters

When a computer application processes a text document, it typically breaks down text into smaller elements that correspond to the smallest unit of data for that process. These units are called text elements. The composition of a text element is dependent on the particular process it undergoes. The Arabic ligature lam-alef is a text element for the rendering process but not for other character operations, such as sorting.

In addition, the same process applied to the same string of text requires different text elements depending on the language associated with the string. Figure 3 shows sorting applied to the string "ch." If this string is part of English text, the text elements for the process of sorting are "c" and "h." In Spanish text, however, the text element for sorting is ch because ch is sorted as if it were a single character.

Figure 3 Text Elements and Collation

Spanish	English
curra	charm
chasquido	current
dano	digit

For other text-processing operations, text elements might constitute units smaller than those traditionally called characters. Examples are the accents and diacritical marks of the Latin script. These small text elements interact graphically with a noncombining character called a base character. The acute accent interacts with the base character A to form the character A acute. If a given font does not have the character A acute, but it does have A and acute accent as separate glyphs, the character A acute has to be divided into smaller units for the rendering process.

In Thai script, vowels and consonants combine graphically so that the vowel mark can be either before, above, below, or after a consonant, thus forming one display unit. This unit becomes the text element for purposes of rendering. For a process such as advance to next character, however, the individual vowels and consonants are the natural units of operation and are therefore the text elements.

There is no simple relationship between text elements and code elements. As we have shown, this relationship varies both with the language of the text and with the operation to be performed by the application. In earlier encoding systems such as ASCII or others with a strong relationship to a language, this problem was not apparent. When designing a universal character code, the Unicode designers acknowledged the issue and analyzed which character elements have to be encoded as code elements to represent the scripts of Unicode across multiple languages. Rather than burden the character code with the complexity of encoding a rich set of text elements, the Unicode Technical Committee decided that the mapping of code elements to more complex text elements should be performed at the application level.

Code Space Structure

The Unicode code space is the full 16-bit space, allowing for 65,536 different character codes. As shown in Figure 4, approximately 50 percent of this space is allocated. This code space is logically divided into four different regions or zones.

[Figure 4 (Code Space Allocation for Scripts) is not available in ASCII format.]

The A-zone, or alphabetic zone, contains the alphabetic scripts. The first 256 positions in the A-zone are occupied by the ISO 8859-1, or 8-bit ANSI codes, in such a way that an 8-bit ASCII code maps to the corresponding 16-bit Unicode character through padding it with one null byte. The positions corresponding to the 32 ASCII control codes 0 to 31 are empty, as well as the positions 0x0080 to 0x009F.

The characters of other alphabetic scripts occupy code space in the range from 0x0000 to 0x2000. Not all of the space is currently occupied, leaving room to encode more alphabetic scripts.

The remainder of the A-zone up to 0x4000 is allocated for general symbols and the phonetic (i.e., nonideographic) characters in use in the Chinese, Japanese, and Korean languages.

The second zone up to 0xA000 is the ideograph, or I-zone, which contains the unified Han characters. Currently about 21,000 positions have been filled, leaving virtually no room for expansion in the I-zone.

The third zone, or O-zone, is a currently unallocated space of 16K. Although several uses for this space have been proposed, its most natural use seems to be for more ideographic characters. However, even 16K can hold only a subset of the ideographic characters.

The fourth zone, the restricted or R-zone, has some space reserved for user-defined characters. It also contains the area of codes that are defined for compatibility with other standards and are not allocated elsewhere.

PROCESSING UNICODE TEXT

The simplest form of Unicode text is often called plain Unicode. It is a text stream of pure Unicode characters without additional formatting or attribute data embedded in the text stream. In this section, we discuss the issues any application faces when processing such text. Processing in this context applies to the steps such as parsing, analyzing, and transforming that an application performs to execute its required task. In most cases, the text processing can be divided into a number of primitive processing operations that are typically offered as a toolkit service on a system. In describing Unicode text processing, we discuss some of these primitives.

Code Conversion

One of the goals of Unicode is to make it possible to write applications that are capable of handling the text of many writing systems. Such an application would typically apply a model that uses Unicode as its native process code. The application could then be written in terms of text operations on Unicode data, which does not vary across the different writing systems.

Today, and for some time to come, however, the data that the application has to process is typically encoded in some code other than Unicode. A frequent operation to be performed is therefore the conversion from the code (file code) in which data is presented to Unicode and back.

One of the design goals of Unicode was to allow compatibility with existing data through round-trip conversion without loss of information. It was not a goal to be able to convert the codes of other character sets to Unicode by simply adding an offset. This would violate the principle of uniqueness, since many characters are duplicated in the various character sets. Most existing character sets therefore have to be mapped through a table lookup. These mapping tables are currently being collected by the Unicode Consortium and will be made available to the public.

It was, however, decided that the 8-bit ASCII, or ISO 8859-1 character set, was to be mapped into the first 256 positions of Unicode. Other character sets (or subsets), such as the Thai standard TIS 620-2529, could also be mapped directly, since character uniqueness was preserved. Also, one of the blocks of Korean syllables is a direct mapping from the Korean standard KSC 5601.

Some character sets contain characters that cannot be assigned code values under the Unicode design rules. Often these characters are different shapes of encoded characters, and encoding them would violate the principle of uniqueness. To allow round-trip conversion for these characters, a special code area, the compatibility zone, was set aside in the R-zone to encode them and to allow interoperation with Unicode. For example, the wide forms of the Latin letters in the Japanese JIS 208 standard were invented to simplify rendering on monospacing terminals and printers.

Character Transformations

A frequently used operation in text processing is the transformation of one character into another character. For example, Latin lowercase characters are often transformed into uppercase characters to execute a case-insensitive search. In most traditional character sets, this operation would translate one code value to another. Thus, the output string of the operation would have the same number of code values as the input string, and both strings would have the same length. This assumption is no longer true in the case of Unicode strings.

Consider the Unicode characters, Latin small letter a + combining grave accent, i.e., a string of two Unicode characters. If this string were part of a French text (in France), transforming a to A would result in one Unicode character, Latin capital letter A. If the same string were part of a French Canadian text, the accent would be retained on the uppercase character. We can therefore make two observations: (1) The string resulting from a character transformation may contain a different number of characters than the original string and (2) The result depends on other attributes of the string, in this case the language/region attribute.

Another important character transformation operation is a normalization transformation. This operation transforms a string into either the most uncomposed or the most precomposed form of Unicode characters. As an example, we consider the different spellings of the combination:

Ü

Latin capital letter U
with diaeresis and grave accent

This letter has been encoded in precomposed form in the Additional Extended Latin part of Unicode. There are two additional spellings possible to encode the same character shape:

Ü +

Latin capital letter U with diaeresis
+ combining grave accent

and

U + .. + `

Latin capital letter U
+ combining diaeresis
+ combining grave accent

The most uncomposed and the most precomposed forms of these spellings can be considered normalized forms. When processing Unicode text, an application would typically transform the character strings into either of these two forms for further processing.

Note that the spellings:

Ù + ..

Latin capital letter U
with grave accent
+ combining diaeresis

and

U + ` + ..

Latin capital letter U
+ combining grave accent
+ combining diaeresis

would result in a different character:

[Note: The resulting character is not readable in ASCII formatted text. The character should be Latin capital U with a grave accent above it and a diaeresis above that.]

This result is due to the rule that diacritical marks, which stack, must be ordered from the base character outwards.

Byte Ordering

Traditional character set encodings, which are conformant to ISO 2022 and the C language multibyte model, consider characters to be a stream of bytes, including cases in which a character consists of more than one byte. Unicode characters are 16-bit entities; the standard does not make any explicit statement about the order in which the two bytes of the 16-bit characters are transmitted when the data is serialized as a stream of bytes.

The ordering of bytes becomes an issue when machines with different internal byte-order architecture communicate. The two possible byte orders are often called little endian and big endian. In a little-endian machine, a 16-bit word is addressed as two consecutive bytes, with the low-order byte being the first byte; in a big-endian machine, the high-order byte is first. Today all computers based on the Intel 80x86 chips, as well as Digital's VAX and Alpha AXP systems, implement a little-endian architecture, whereas machines built on Motorola's 680xx, as well as the reduced instruction set computers (RISC) of Sun, Hewlett-Packard, and IBM, implement a big-endian architecture. In blind interchange between systems of possibly different byte order, Unicode-encoded text may be read incorrectly. To avoid such a situation, Unicode has implemented a byte-order mark that behaves as a signature. As shown in Figure 5, the byte-order mark has the code value 0xFEFF. It is defined as a zero-width, no-break space character with no semantic meaning other than byte-order mark.

[Figure 5 (Byte-order Mark) is not available in ASCII format.]

The code value corresponding to the byte-inverted form of this character, namely 0xFFFE, is an illegal Unicode value. If the byte-order mark is inserted into a serialized data stream and is read by a machine with a different byte-order architecture, it appears as 0xFFFE. This fact signals to the application that the bytes of the data stream have been read in reverse order from that in which they were written and should be inverted. Applications are encouraged to use the byte-order mark as the first character of any data written to a storage medium or transmitted over a network.

Display of Bidirectional Strings

To facilitate internal text processing, a Unicode-compliant application always stores characters in logical order, that is, in the order a human being would type or write them. This causes complications in rendering when text normally displayed right to left (RL) is mixed with text displayed left to right (LR). Hebrew or Arabic is written right to left, but may contain characters written left to right, if either language is mixed with Latin characters. Numerals or punctuation mixed with Hebrew or Arabic can be written in either order.

The Default Bidirectional Algorithm

Unicode defines a default algorithm for displaying such text based on the direction attributes of characters. We outline the algorithm in this paper; for details, see both volumes of the Unicode standard.[4,5] (It is important to consult the second volume because it contains corrections to the algorithm given in the first volume.)

All printing characters are classified as strongly LR, weakly LR, strongly RL, weakly RL, or neutral. In addition, Unicode defines the concept of a global direction associated with a block of text. A block is approximately equivalent to a paragraph. The first task of the rendering software is to determine the global direction, which becomes the default. Embedded strings of characters from other scripts are rendered according to their direction attribute. Neutral characters take on the attribute of surrounding characters and are rendered accordingly.

Directionality Control

Although the default algorithm gives correct rendering in most realistic cases, extra information occasionally is needed to indicate the correct rendering order. Therefore, Unicode includes a number of implicit and explicit formatting codes to allow for the embedding of bidirectional text:

Left-to-right mark	(LRM)
Right-to-left mark	(RLM)
Right-to-left embedding	(RLE)
Left-to-right embedding	(LRE)
Left-to-right override	(LRO)
Right-to-left override	(RLO)
Pop directional formatting	(PDF)

It must be pointed out that the directional codes are to be interpreted only in the case of horizontal text and ignored for any operation other than bidirectional processing. In particular, they must not be included in compare string operations.

The LRM and RLM characters are nondisplayable characters with strong directionality attributes. Since characters with weak or neutral directionality take their rendition directionality from the surrounding characters, LRM and RLM are used to influence the directionality of neighboring characters.

The RLE and LRE embedding characters and the LRO and RLO override characters introduce substrings with respect to directionality. The override characters enforce a directionality and are used to enforce rendering of, for instance, Latin letters or numbers from right to left. Substrings can be nested, and conforming applications must support 15 levels of nesting. Each RLE, LRE, LRO, or RLO character introduces a new sublevel, and the next following PDF character returns to the previous level. The directionality of the uppermost level is implicit or determined by the application.

Only correct resolution of directionality nesting gives the correct result. In general it cannot be assumed that a string of text that is inserted into other bidirectional text will have the correct directionality attributes without special processing. This may result in the removal of directional codes in the text or in the addition of further controls. As shown in Figure 6,

particular care needs to be taken for cut-and-paste operations of bidirectional text.

[Figure 6 (Cut and Paste of Bidirectional Text) is not available in ASCII format.]

Transmission over 8-bit Channels

Existing communication systems often require that data adheres to the rules of ISO/IEC 2022, which reserve the 8-bit code values between 0x00 and 0x1F (the C0 space), between 0x80 and 0x9F (the C1 space), and the code position DELETE.[1] Since Unicode uses these values to encode characters, direct transmission of Unicode data over such transmission systems is not possible.

The Unicode designers, in collaboration with ISO, have therefore proposed an algorithm that transforms Unicode characters so that the C0 and C1 characters and DELETE are avoided. This algorithm, the UCS transformation format (UTF), is part of the ISO 10646 standard as an informative annex. It is expected that it will be included in the revised Unicode standard.

The transformation algorithm has been conceived in such a way that the characters corresponding to the 7-bit ASCII codes and the C1 codes are represented by one byte (see Figure 4). Code positions 0x00A0 through 0x4015 (which include the remainder of the extended Latin alphabet) are represented by two bytes each, and three bytes each are used for the remaining code values.

Originally, UTF had been proposed for use in data transmission and to avoid the problem that embedded zero bytes represent for C language character strings in the char data type. Subsequently, it has been proposed to use UTF in historical operating systems (e.g., UNIX) to store Unicode-encoded system resources such as file names.[10]

Modifications of UTF have therefore been proposed to address other special requirements such as preservation of the slash (/) character.[11] It remains to be seen which of these various transformation methods will be widely adopted.

Handling of Combining Characters

In some of the operations discussed above, we have indicated that the presence of combining characters requires processing Unicode text differently from text encoded in a character set without combining characters. Normalization or transformation of the characters into a normalized form is usually a first helpful step for further processing. For example, to prepare a text for a comparison operation, one may wish to decompose any precomposed characters. In this way, multiple-pass comparison and sorting algorithms, which typically pass through a level that ignores diacritical marks, can be applied almost unchanged.[12]

For simple comparison operations, the application must decide on a policy of what constitutes equality of two strings. If the string contains characters with a single diacritical mark, it can choose either strong matching, which requires the diacritical marks in both strings, or weak matching, which ignores diacritical marks. If the text includes characters with more than one diacritical mark for a medium-strong match, the presence of certain marks might be required but not of others. Strong matching is required for the Greek word for micromaterial mikroüliká and the Greek diminuitive form of small mikroúlika. Without the diacritical marks, the words would be identical.

Unicode requires that combining characters follow the base character. This solution was chosen over the alternatives of (1) precede and (2) precede and follow, for various reasons.[13] Text-editing operations must take into account the presence and ordering of diacritical marks. A user-friendly application should be consistent in its choice of text element on which operations such as next character or delete character operate. This choice should feel natural to the user. For example, in Latin, Greek, and Cyrillic, the expectation would be that accented characters are the unit of operation, whereas in Devanagari and Thai, where several combining characters and a base character combine into a cell, the natural unit is the individual character.

IMPLEMENTATION ISSUES

In this section we describe some of the approaches that can be taken to support Unicode. As a concrete example, we describe how the Microsoft Windows NT operating system uses Unicode as the native text encoding and maintains compatibility with existing applications based on a different encoding.

General Considerations in Adding Unicode Support

Informal discussions with vendors planning to support Unicode indicate that the following data types and data access are being considered when using the C programming language.

1. A new data type would be designated for Unicode only. It would be directly accessible by the application, e.g., typedef unsigned short UNICHAR.

   The Unicode-only data type has the advantage of being unencumbered with preconceptions about semantics or usage. Also, since the application knows that the contents are in Unicode, it can write code-set-dependent applications.

   The major disadvantage is that the data type would vary from one vendor or platform to another and would

therefore have no standard string-processing libraries.

2. An existing data type, such as wchar_t in C would be used. (Note that the char data type is appropriate only if char is defined as 16 bits, or if the string is given some further structure to define its length by means other than null termination. Similar issues may exist in other languages.)

   The use of an existing data type has the advantage of being widely known and implemented; however, it also has the disadvantage of preexisting assumptions about behavior and/or semantics.

3. An opaque object would be used. Since the data in these objects is not visible to the calling program, it can only be processed by routines or by invoking its member functions (e.g., in C++).

   Use of an opaque object has the advantage of hiding much of the complexity inherent in the world's writing systems from the application writer. It has the disadvantages common to object-oriented systems, such as the need for software engineers to learn a new programming paradigm and a set of class libraries for the Unicode objects.

How Windows NT Implements Unicode

The Windows NT design team started with several goals to make an operating system that would preserve the investment of customers and developers. These goals affected their decisions regarding the data types and migration strategies described in the previous section.

The goals related to text processing were to

1. Provide backward compatibility

   a) Support existing MS-DOS and 16-bit MS Windows applications, including those based on 8-bit and double-byte character set (DBCS) code pages.

   b) Support the DOS file allocation table file system.

2. Provide worldwide character support in

   a) File names

   b) File contents

   c) User names

As described later in this section, these conflicting goals were

met under a single Windows NT architecture, if not simultaneously in the same application and file system, then by clever segregation of Windows NT into multitasking subsystems. These goals also affect the way Microsoft recommends developers migrate their existing applications to Windows NT.

The Basic Approach. Microsoft's overall approach is close to that of using a standard data type that accesses data mainly through string-processing functions. In addition, Microsoft defined a special set of symbols and macros for application developers who wish to continue to develop applications based on DOS (e.g., to sell to those with 286 and 386SX systems), while they migrate their products to run as native Win32 applications on Windows NT. The developer can then compile the application with or without the compiler switch -DUNICODE to produce an object module compiled for a native Windows NT or a DOS operating environment, respectively.

Dual-path Data Types. To select the appropriate compilation path, Microsoft provides C language header files that conditionally define data types, macros, and function names for either Unicode or traditional 8-bit (and DBCS) support, depending on whether or not the symbol UNICODE has been defined. An example of a data type that illustrates this approach is TCHAR. If UNICODE is defined, TCHAR is equivalent to wchar_t. Otherwise, it is the same as char. The application writer is asked to convert all instances of char to TCHAR to implement the dual development strategy.

String-handling Functions. Similarly, the macro TEXT is defined to indicate that string constants are wide string constants when UNICODE is defined, or ordinary string constants otherwise. Application writers should surround all instances of a string or character constant with this macro. Thus, "Filename" becomes TEXT("Filename"), and 'Z' becomes TEXT('Z'). The compiler treats these as a wide string or character constant if UNICODE is defined, and as a standard char based string or character otherwise.

Finally, there are symbol names for each of the various string-processing functions. For example, if UNICODE is defined, the function symbol name _tcscmp is replaced by wcscmp by the C preprocessor, indicating that the wide character function of that name is to be called. Otherwise, _tcscmp is replaced with the standard C library function strcmp. Details of this procedure can be found in Win32 Application Programming Interface.[14]

Procedures for Developing/Migrating Applications in the Dual Path. In his paper on "Program Migration to Unicode," Asmus Freytag of Microsoft explains the steps used to convert an existing application to work in Unicode and retain the ability to compile it as a DOS or 16-bit Windows application.[15] The basic

idea is to remove the assumptions about how a string is represented or processed. All references to string-related objects (e.g., char data types), string constants, and string-processing functions are replaced with their dual-path equivalents. The following steps are then taken.

1. Replace all instances of char with TCHAR, char* with LPSTR, etc. (For a complete listing, see "Program Migration to Unicode.")[15]

2. Replace all instances of string or character constants with the equivalent using the TEXT macro.[16] For example,

```
char filemessage[] = "Filename";
char yeschar = 'Y';
```

becomes

```
TCHAR filemessage[] = TEXT("Filename");
TCHAR yeschar = TEXT('Y');
```

3. Replace standard char based string-processing functions with the Win32 functions. (See page 221 of Win32 Application Programming Interface, for a complete listing.)[14]

4. Normalize string-length computations using sizeof() where appropriate. For example, direct computation using address arithmetic should take the form: string_length = (last_address -- first_address)*sizeof(TCHAR);

5. Mark all files with the byte-order mark.[17]

6. Make other, more substantial changes.

Most character-code-dependent processing should be taken care of by step 3, assuming the developer has used standard functions. If the source code makes assumptions about the encoding, it will have to be replaced with a neutral function call. For example, the well-known uppercasing sequence

```
char_upper = char_lower + 'a' -- 'A';
```

implicitly assumes the language and the uppercasing rules are English. These must be replaced with a function call that accesses the Windows NT Natural Language Services.

SUMMARY

A universal character encoding -- the Unicode standard -- has been developed to produce international software and to process and render data in most of the world's languages. The standard, often referred to as Unicode/10646, was jointly developed by vendors and individual experts and by the International Organization for Standardization and International Electrotechnical Commission (ISO/IEC). Unicode breaks the (incorrect) principle that one character equals one byte equals one glyph. It stipulates the use of text elements that are dependent on the particular text operation. A number of software vendors are now moving to support Unicode. Microsoft's implementation supports Unicode as the native text encoding in its Windows NT operating system. At the same time, it maintains compatibility with existing applications based on 8-bit encoding.

ACKNOWLEDGMENTS

The authors would like to express their thanks to Asmus Freytag of Microsoft Corporation and Masami Hasegawa (ISO/IEC 10646 editor) for their efforts in reviewing this paper.

REFERENCES AND NOTES

1. Information Processing -- ISO 7-bit and 8-bit Coded Character Sets -- Code Extension Techniques, International Standard, ISO 2022:1986 (Geneva: International Organization for Standardization, 1986).

2. Information Technology -- Multiple-Octet Coded Character Set, Draft International Standard, ISO/IEC DIS 10646:1990 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1990).

3. J. Becker, "Multilingual Word Processing," Scientific American, vol. 251 (July 1984): 96-107.

4. The Unicode Standard, Version 1.0, Volume 1 (Reading, MA: Addison-Wesley Publishing Company, 1991).

5. The Unicode Standard, Version 1.0, Volume 2 (Reading, MA: Addison-Wesley Publishing Company, 1992).

6. Information Technology -- Universal Multiple-Octet Coded Character Set (UCS), Draft International Standard, ISO/IEC DIS 10646-1.2:1991 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1991).

7. Unicode 1.0.1 Errata Insert for The Unicode Standard, Version 1.0, Volume 1 and Volume 2 (Reading, MA:

Addison-Wesley Publishing Company, 1992).

8. ISO/IEC 10646 restricts the use of combining characters. See the definitions of level 2 and level 3 in the section Relationship between Unicode and ISO/IEC 10646.

9. Some of the presentation variants are encoded for compatibility with existing standards. For a discussion, see the section Code Conversion.

10. R. Pike and K. Thompson, "Hello World," Usenix Conference, 1993.

11. File System Safe -- UCS Transformation Format (Reading: X/Open Company Limited, 1993).

12. A. LaBonté, "Multiscript Ordering for Unicode," Proceedings of the Fourth Unicode Implementors Workshop, Sulzbach (Unicode Inc., 1992).

13. Private communication, Joseph D. Becker, 1993.

14. Win32 Application Programming Interface (Redmond: Microsoft Press, 1992).

15. A. Freytag, "Program Migration to Unicode," Proceedings of the Second Unicode Implementors Workshop, Merrimack (Unicode Inc., 1992).

16. String constants in source code should be avoided in all cases. They violate one of the fundamental design rules of software internationalization, i.e., that objects dependent on language and/or culture should be isolated into easily accessible modules for the purpose of localization.

17. Unicode defined the code value 0xFEFF to have the semantic byte-order mark (BOM) and encourages software developers to place it as the first character in a Unicode file. (For details, see the section Byte Ordering.)

TRADEMARKS

Win32 and Windows NT are trademarks of Microsoft Corporation.

Motorola is a registered trademark of Motorola, Inc.

Unicode is a trademark of Unicode Inc.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

WordPerfect is a trademark of WordPerfect Corporation.

X/Open is a trademark of X/Open Company Limited.

BIOGRAPHIES

Jürgen Bettels Jürgen Bettels is an internationalization architect and the standards manager for the International Systems Engineering Group. Since 1986, he has worked on many internationalization architectures starting with DECwindows. He participated in the Unicode consortium, ECMA, and X/Open on internationalization. He contributed to the ISO/IEC WG2/SC2, whose work merged Unicode and ISO 10646 into a single universal character encoding. Prior to joining Digital, he was a physicist at the European particle laboratory, CERN. Jürgen has the degree of Diplom Physiker (physicist) from the University of Aachen.

F. Avery Bishop Avery Bishop is the program manager for Windows NT/Alpha internationalization. Prior to this position, he worked in ISE as Digital's representative to the Unicode consortium and the ANSI X3L2 technical advisory group on character encoding. He worked with ISO/IEC WG2/SC2, Unicode, and others in Digital to merge Unicode and ISO 10646 into a single universal character encoding. Prior to that, he managed projects at DECwest and worked as the product management manager for ISE in Japan. Avery has a Ph.D. in electrical engineering from the University of Utah.