International Support in Applications and Systems Software

Globalisation and localisation are two capabilities required of modern applications

Mark Davis & Jack Grimes

THE SCOPE OF THE INTERNATIONALISATION PROBLEM

The goal for Taligent and for many other organisations is to ensure that all programs are usable all over the world. These are called *global programs* and have a single binary form that is used everywhere. A global program can be "localised" for use in some region, usually a country.

Today, most programs are localised without being global in the first place, leading to many versions of the program—potentially one for each region. Not only is it expensive to create all these versions, but it also means that a document created with a particular localised version of a program will display incorrectly when opened with another version of the program. Typically you need a knowledgeable, native speaker to complete a successful localisation. Glossaries are created and used to make the translation consistent. For example, in many languages the word for mouse and rat are the same, so one must be careful not to simply translate English words into another language to refer to a particular piece of equipment. The same terms should be used across localised applications, e.g., two programs should use the same terms for "click on a menu item."

The goal of a single, global program has two aspects:

- Can you enter all the data that program uses in your native language? For example, can I type in Japanese anywhere where text is needed?
- Is the presentation of that program, e.g., text and graphics on the screen, in your native language? Can I look at the presentation of that program and understand all the menus, diagrams, graphics, icons, or any of the elements that help me see how to use the program?

SO, WHAT HAPPENED TO ASCII?

In the past, many programs were based on ASCII, period. One fixed character set. One size fits all. To overcome limitations, programs were designed based on ASCII with variations that allowed the programmer to support several European languages. These used variations, e.g., where the code for a character like "[" was used for an umlaut, "ö." Many of these seven-bit ASCII systems are in common use today.

Later, standard character sets were developed, like the ISO 8859 series, that used standard ASCII for the first seven bits and then had 128 more characters added, forming a family of eight-bit extended ASCII character sets. This worked well for each of several European languages, but there are still two major problems:

- First, while one of these ISO 8859 standards might work for an individual European language, it doesn't work for combinations. If you have a company that is exchanging data among Germany, Iceland, and Poland, there isn't a single character set that covers all the required characters.

  Extended ASCII provided a national solution in some cases. However, its time has passed. Commerce is increasingly becoming international, both between organisations and within multinational companies.

  To overcome this problem, the next enhancement was to embed switching codes (called *introducers*) in the text strings that would allow one to switch in and out of different eight-bit ASCII character sets within the document. This approach works after a fashion, but is extremely clumsy for a developer to deal with. First, it requires agreements on both ends as to the meanings of various switching codes. Second, text parsing is complicated because you always need to know what character set is active. This is difficult when the user positions the text cursor at some arbitrary point in the text. Third, search-and-replace produces false matches because the matching is done on the character codes, not the font information. The algorithm might replace Greek letters that were not part of the text string. This is one more thing for the programmer to anticipate specifically. Most of the unexpected behaviour can be detected and programmed around, but the code becomes increasingly complex.
- A second major problem exists, even with the addition of introducers. The eight-bit extended ASCII sets are not large enough to handle the character sets for Japanese or other eastern languages. For these markets, double-byte character (DBC) sets were developed to handle languages that require thousands or tens of thousands of characters.

  Typically, these large character sets are combinations of single- and double-byte encodings, including introducers such as

Illustration: ©John Dykes/SIS

shift-JIS, which are embedded in the string. An additional complication for the developer in dealing with the DBC encoding is that the code is different from the code that deals with the single-byte sets. From the programmer's perspective, the double-byte sets behave in a different way. A wide variety of solutions were created, most of which were, and still are, incompatible.

LOCALES

A locale is a set of information about how to deal with a country or region. A region can be a part of a country that has different languages within its borders, e.g., Switzerland or Belgium. A locale lets you set the localisation language, and if the system has the proper support, you can establish a format for numbers, dates, and currency. Then numbers will be displayed in, for example, the French format. The locales affect all language-related input and output, as they are part of the application environment and affect how everything works.

Programmers have tried to solve the locale problem by using limited character sets like ASCII or extended ASCII and by adding introducers. This works fairly well within a single region for one language, but it doesn't work when you are handling multiple languages inside the same document. Furthermore, only one part of a locale is the character set. Time zones and number representations also can change. With a globalised product, the localisation is greatly simplified and a program like a word processor can be localised with a few days or weeks of work by one native speaker.

Finally, the programmer has to deal with the character set issue at the very bottom of the text system where it is the most difficult, since it is used for both the input of data and in the presentation of strings. In addition, virtually every application must deal with text, so this problem must be handled over and over, by each developer for each program. Sounds like a good candidate for systemwide support!

A THIRD PROBLEM HAS GONE AWAY

There was a third problem that has largely been solved by modern systems. Originally when developers tried to internationalize programs, they put the language-specific text strings in their source program as literal strings—so-called hard-coded. Then, to modify these presentation strings, they modified the source. This caused problems for developers who wanted to keep their source code proprietary.

Today, most systems provide for the storing of these language-specific text strings in a separate place where a different developer, usually in the destination country, can modify the language-specific text strings without needing access to the source code. These are usually called *message files* or *resources*.

SOME LANGUAGES ARE HARDER TO DISPLAY THAN OTHERS

Traditionally the programmer thought of characters as units that sequenced across the page. Further, each character fitted into a box and the sequence was laid out by putting the boxes next to each other. As you encounter a character in a byte string, you put it on the screen, and that's that. In fact, developers didn't distinguish between a character and its visual presentation on paper or on a screen. This doesn't work for languages like Hebrew or Arabic, which are written cursively (the letters in a word are all joined together as in handwriting). Here, an individual character shape may need to change at its beginning or end as typing proceeds because each character's shape is affected by the surrounding characters. No matter what the language, characters and their visual representation should not be treated as interchangeable concepts, especially if you are creating global programs.

CHARACTERS VERSUS GLYPHS

Characters and glyphs are two different things. A character represents a specific meaning (semantic), while a glyph provides a graphical representation of a character. A character's glyph can change shape dramatically and is not necessarily tied one-to-one with the character (see Table 1). The character *a* is defined as the meaning of the character, not its appearance. An *a* is an *A* is an A! Further, a single glyph can represent two characters, e.g., *Æ* representing the *A, E* sequence in some cases or the printer's use of a single glyph for the *f* sequence in the word *difficult*. There are other cases where you have two glyphs representing one character. For example, in South Indian languages a single character sometimes separates into two pieces that are placed around a preceding character.

In summary, each character has unique semantics but can have its own glyph, can combine with one or more characters to create one glyph, or can separate into multiple glyphs. The character has semantics, an A means an A. However, the appearance of an A can vary widely, e.g., by style, as in an *italics A*; by font, as in a Helvetica A; or by context, as in a cursive script. The same glyph may be used for different characters, e.g., a hyphen and an en dash may have the same glyph, or the Greek letter capital rho may have the same glyph as a capital *P* in English. This all gets fairly complicated, but such are the requirements for international software.

Glyph (presentation)	Character (semantics)	Comment
a, A, *A*	letter a	Each character can have multiple glyphs, at different times
Æ	A followed by an E	Two or more characters can have one glyph
P, ρ	English letter P Greek letter rho	The same glyph can represent more than one character at different times
None	ASCII "bell"	Some characters (e.g., control characters) don't have glyphs
See text for example	Some South Indian characters	One character can have two glyphs at the same time

Table 1. Characters (semantics) and glyphs (presentations) are different concepts and don't necessarily map one-to-one.

ENTER UNICODE

An important part of the solution to this character set problem from a global perspective is the Unicode Consortium (see sidebar) and their efforts at compatibility with an international standard for character encodings, ISO/IEC 10646.[1,2]

Taligent has long been a strong advocate of Unicode and has supported the Unicode Consortium since its inception. Unicode provides a single, 16-bit, global character set. This is crucial for creating globally distributable software. There are no introducers or strings of data tagged with character set information, and the character set doesn't vary with locale.

NO PERSON IS AN ISLAND

You might think that if you live in only one locale, or don't go out-

Arabic	Greek	Kana	Tamil
Armenian	Gujarati	Kannada	Telugu
Bengali	Gurmukhi	Lao	Thai
Cyrillic	Han	Latin	Zhuyinfahao
Devanagari	Hangul	Malayalam	
Georgian	Hebrew	Oriya	

Table 2. Major scripts in use in the world today whose characters are provided by the Unicode standard

side your house, all this fuss might be ignored. However, if you look at the number of symbols that are used in newspapers or books, even within Roman languages, it is greater than 256, the limit of an eight-bit extended ASCII set. Often an alternate font is used. For example, in this article's summary the arrows are set in Zapf Dingbats. Different fonts simply mean that the same character codes map to entirely different glyphs. These different glyphs should have different character codes, as they do in Unicode. When this document was sent through email to the editor, information was lost. Even on a single system in a single language you need a character set larger than 256 characters.

In addition to losing information, you also lose legibility. Back in the old days we had five-bit codes for teletypes that only supported uppercase letters and a few symbols. Sure, we got by, but legibility suffered. No one wants to return to that code set!

SUMMARY

ASCII and extended ASCII have served the computing industry for many years, but no longer provide the necessary character set facilities for modern organisations that must function well in an international setting.

- *Globalisation* is the process of designing a program or system so that a single binary version can be sold and used in any region of the world.
- *Localisation* is taking the program or system and translating the text strings and other visual elements so they can be used in a locale, i.e., a region or country.

If the developer has created a global product, the user types in the data in his or her language using the appropriate locale. Selecting a locale also changes the presentation to the user's language. Both of these capabilities are expected of modern applications.

References

1. The Unicode Consortium. THE UNICODE STANDARD: WORLDWIDE CHARACTER ENCODING, VERSION 1.0 (2 volumes), Addison-Wesley, Reading, MA, 1991.
2. The Unicode Consortium. THE UNICODE STANDARD, VERSION 1.1, Unicode Technical Report #4 (prepublication ed.), The Unicode Consortium, Mountain View, CA, 1993.

Mark Davis and Jack Grimes are with Taligent, Inc., 10201 N. De Anza Blvd., Cupertino, CA 95014; email: Mark_Davis@taligent.com and jgrimes@taligent.com.

Unicode, a fixed-width, 16-bit character-encoding system, contains codes for every character needed by the major writing systems in use throughout the world today. Unicode provides full character coverage for the major scripts, listed in Table 2, as well as punctuation, symbols, and control characters. The character set for each script is independent, i.e., even if a character appears in multiple scripts, it has a separate code within each script. For example, the character A has a code for the Roman alphabet and another code for the Greek alphabet. Note that this applies to scripts, not languages. The character A is identical for the English and French languages, for example. In all, the Unicode standard provides codes for over 34,000 characters from the world's alphabets, ideograph sets, and symbol sets. There are over 24,000 unused codes for expansion, and over 6000 codes are reserved for private use by software and hardware developers.

In addition to the script name, Unicode associates other semantic information with each character. Each character has type properties that describe the usage of the character. Some examples of type properties are

- **uppercase, lowercase, and uncased letters**
- **diacritical marks**
- **characters used to represent digits**
- **punctuation marks**
- **symbols**
- **control characters**

The Unicode standard follows a set of fundamental principles:

- **Provides a simple and consistent interface. It uses fixed-width 16-bit character codes and does not depend on states or modes.**
- **Incorporates character sets from many existing standards, e.g., it includes the Latin-1 character set as its first 256 characters. It includes the repertoire of characters from many other international, national, and corporate standards as well.**
- **Consolidates Chinese, Korean, and Japanese ideographs by assigning a single code for each ideograph that is common to more than one of these languages.**
- **Allows marked-character creation through character composition. It encodes each character and diacritic or vowel mark separately and allows the characters to be combined to create a marked character (such as é). It also provides single codes for marked characters to comply with preexisting standards.**

The Unicode Consortium, a nonprofit organisation, was founded in 1991. Members of the consortium include major computer corporations, software producers, database vendors, research institutions, international agencies, and various user groups. Full members of the Consortium are Apple Computer, Inc.; Digital Equipment Corporation; Hewlett-Packard Company; IBM Corporation; Lotus Development Corporation; Microsoft Corporation; NeXT Computer, Inc.; Novell, Inc.; The Research Libraries Group, Inc.; Symantec Corporation; Taligent, Inc.; Unisys Corporation; and WordPerfect Corporation.

For details on Unicode design and usage, see THE UNICODE STANDARD 1.0[1] and Unicode Technical Report #4.[2] Unicode Technical Report #4 describes the amendments to Unicode 1.0 that constitute Unicode 1.1, the version of Unicode aligned with ISO/IEC 10646 and implemented by the CommonPoint application system.