



## Building International Software

*By Herbert Tang*

[More on internationalization](#)

Package developers often concentrate first on getting the software to work for their target market, deferring the question of adapting it to other national languages. Once the package is up and running, and revenue is coming in, there will be time to think about making the package accessible to international customers. This may be a serious mistake. By following a few simple rules, it is possible to ensure that software can be readily adapted to any national language. These rules add little cost to the initial development. Retrofitting the code after the fact is far more expensive.

This report outlines how to build software that is ready for internationalization from the start. The package must be built so it can support character sets other than Latin and can correctly manipulate text in these character sets, including correct sorting, comparison, string management and display. The code must support non-Gregorian dates and different number display formats. Finally, it must be practical to translate the text accurately and insert the translation into the software.

It is beyond the scope of this paper to give rules for every programming language or development tool that may be used to build a package. However, the general guidelines given here can be easily adapted into specific rules for the chosen technology.

Before getting into the guidelines, this report briefly discusses international writing systems and the Unicode representation of these systems. Although handling text is not the only concern in making code international, it is by far the most important. It is essential to have a solid understanding of writing systems and Unicode.

### Writing Systems

A writing system records the spoken word in visual format. Modern writing systems fall into four broad categories: Abjad, Alphabetic and Syllabic systems, derived from early Semitic scripts, and Logographic systems, based on classical Chinese script.

In the abjad, alphabetic and syllabic systems, a number of characters combine to spell a word. Spaces, commas or other punctuation marks separate words. Alphabetic systems are normally written in horizontal rows, with rows arranged top to bottom. Systems derived from Brahmi and Greek are usually written left to right, while systems derived from Aramaic are usually written right to left.

Modern logographic writing systems are based on classical Chinese, still used in Taiwan and Singapore. They include simplified Chinese (used in the People's Republic of China) and Japanese. Symbols are built up from a small set of strokes, with anything from one to 48 strokes making up a symbol. There are no spaces between symbols. Traditionally

the East Asian logographic systems are written in vertical rows, top to bottom, with rows arranged left to right, but horizontal rows with symbols written left to right and rows arranged top to bottom are acceptable and are widely used.

When a script is used for different languages, accents are often added to represent sounds unique to each language. For example, the Latin character "a" may be accented in different European languages as à, á, â, ã, ä or å. Additional dots and squiggles may be added to standard Arabic characters when writing other languages such as Farsi. The accent characters usually have no effect on dictionary sequence.

## **Computer Representations of Writing**

The method of representing writing in computer systems continues to evolve. The most recent standard - Unicode - promises to be more long-lived than its predecessors, but is only a partial standard. Sequencing and visual representation are not addressed.

### **ASCII**

The earliest computer systems only supported the Latin alphabet, with 128 characters in the 7-bit ASCII format. Most computers from the 1970's onward used the 8-bit Latin-1 standard with 256 characters, the first 128 the same as the earlier 7-bit standard. The characters added in the 8-bit standard include additional punctuation symbols, currency symbols and accented European characters. For obscure reasons, many mail gateways still only support the 7-bit standard.

Latin-1 coding was good enough for Latin alphabets in the days of green screens and impact printers. Variants could be used for Greek, Arabic or other alphabets as long as the terminals and printers were configured to display the appropriate character in place of the Latin character and, with Arabic, configured to present the characters "back to front". But the ASCII and Latin-1 coding schemes cannot handle East Asian character sets, which have thousands of different symbols.

### **Double-Byte**

The first efforts to handle East Asian character sets were based on "double byte" coding schemes, in which two internal characters are used to represent one external character. Two bytes in 8-bit encoding support 65,536 characters, enough to assign a unique value to every common alphabetic or East Asian character in the world, but not enough to support every obscure character set.

A number of different double byte standards were developed. The most important was ISO 2022, which allowed text to be stored in many different languages, with switches from one language to another handled by announcement of the change. But ISO 2022 is complicated and inefficient. The East Asian countries are abandoning this standard and moving to ISO 10646-1:2000, more fully defined in Unicode 3.0.



## Unicode

Unicode is the international standard for representing all character sets in the world within a single coding scheme. It is closely aligned with the international standard ISO/IEV 10646, also known as the Universal Character Set (UCS). Unicode is supported in most modern environments, but not in Windows 95/98. This paper assumes that new packages need not run on the older versions of Windows.

With Unicode, every character is assigned a unique number. It is thus possible to mix characters from many different writing systems in a single text area, as long as the printer or workstation supports these writing systems. The Basic Multilingual Plane (BMP) holds all characters used in current writing systems, using numbers up to 65536. Higher numbers are used for obsolete systems such as cuneiform or hieroglyphics.

Unicode defines three encoding formats that let the same data be transmitted in 8-bit, 16-bit or 32-bit formats. All three formats encode the same set of characters, and can be efficiently converted from one to another without loss of data.

**UTF-8 (Transitional)** The most efficient format when text is mainly in the Latin alphabet. Characters corresponding to the first 128 ASCII characters have the same encoding as the ASCII characters. Other characters take two or three bytes. UTF-8 takes 50% more space than UTF-16 for East Asian text.

**UTF-16 (UCS-2)** Balances between efficient storage and efficient processing. All the characters in the Basic Multilingual Plane (BMP) - all the characters relevant to any package - use 2 bytes. The rarer characters use 4 bytes.

**UTF-32 (UCS-4)** The simplest but most wasteful of space. All characters are represented in 4 bytes (32 bits.) There is unlikely to be demand for a Hieroglyphic version of a software package, so UTF-32 may be eliminated.

## Which Unicode Format?

The standards bodies see UTF-8 as "transitional", an interim standard to be eventually replaced by the UTF-16 standard. However, UTF-8 will be widely used long into the future. It is the standard method of representing international characters in HTML. This alone will guarantee a long life. The benefits of UTF-8 compared to UTF-16 include:

- UTF-8 uses less space than UTF-16 for alphabetic systems. It uses more space for East Asian symbols, but these are inherently compact: one symbol = one word or syllable. A data element designed to hold a 30-character Latin UTF-8 text string will be able to support 10 syllables of Chinese or Japanese, enough to represent the same concepts. With UTF-16, the data element would have to be twice as long to hold the Latin text, far longer than needed to hold the equivalent East Asian symbols.

- UNIX is based on ASCII. UTF-16 codes contain bytes like '\0' or '/' that have special meaning in file names and other C library function parameters. Most UNIX tools expect ASCII files and cannot read 16-bit words as characters. A decision to use UTF-16 would cause a series of minor problems within the software, which would be costly to correct.

This paper therefore recommends using UTF-8 to hold all text that is visible to the end user. When interfacing with systems that expect UTF-16 (or even UTF-32) it will be a trivial job to convert the text strings to these formats in the interface.

## Defining and Manipulating Text

Software that may be deployed in the international market must store text in the right format, and must follow rules for manipulating text that may not be immediately obvious to someone who is only familiar with the Latin character set. Guidelines for defining and manipulating text follow.

### *Text Representation*

Data elements that hold text visible to the end user - data such as names and addresses, labels on reports or forms, error messages, code value mnemonics, help files and so on - must be defined with a "Unicode" type so they can be manipulated correctly.

Avoid defining short or fixed-length text elements. A single-byte element may be enough to hold a Latin-character code value (e.g. "Y" or "N"), but not long enough to hold the corresponding East Asian characters. Define all text elements as variable, no shorter than four bytes.

### *Text Stringing*

Programmers may be tempted to string bits of text together to form a message. This will often produce bad results, even with Latin character sets. For example, if the Italian words "errato" (incorrect) and "cartella" (folder) were strung together, separated by a space, the result would be "errato cartella". The words are in the wrong sequence and the gender of "errato" does not agree with the gender of "cartella". The correct Italian would be "cartella errata". If the equivalent Chinese words were strung together, the result would incorrectly hold a space between the symbols for "incorrect" and "folder".

Three rules:

1. Try to avoid sticking bits of text together - there is usually an alternative.
2. If you must combine text, think of it as inserting textual data at a defined place in another bit of text, rather than stringing the text together
3. Use a standard subroutine to insert data into text. The subroutine may vary its behavior depending on the national language and character set in effect.



The first version of the subroutine may only handle English or alphabetic text:

```
Result = TextInsert ("English", "Product code ### not found", "ABC")  
Result = "Product code ABC not found"
```

This subroutine can later be adapted to handle the rules for different national languages and character sets without affecting the package logic.

### **Character Counting**

UTF-8 characters are variable-length, from one to three bytes long. Most languages include functions or classes that support UTF-8 string manipulation. If these are used correctly, there should be no problem. But code that assumes that it is dealing with fixed-length characters, one byte per character, will not work. The nth byte in a text string will not necessarily contain the nth character.

### **Code Values**

Logic should never refer directly to specific code values unless these values are purely internal - never displayed or printed. Instead, logic should refer to data elements that hold the code values, where the actual value assigned to the data element can be specified when building the software for a given National language (more on this later.)

### **Sequencing & Comparison**

In alphabetic scripts, the letters have a standard sequence. Words are sequenced by the first letter in the word, then the next letter, and so on. With scripts like Latin and Thai, the leftmost character is the most significant. With scripts like Arabic, the rightmost character is the most significant. In East Asian scripts, characters are ordered partly by the number of strokes that they contain, partly by dominant form.

Although Unicode usually assigns numbers to characters in their dictionary sequence, it does not always do so. Simply sorting text strings on their Unicode numbers will result in incorrect results. For example, the characters "A" and "a" are given different Unicode numbers.

Software that sequences text or compares the sequence of text strings must follow the correct rules for Unicode sequencing. For example, comparison logic in Oracle must use the NLSSORT function:

Wrong: `SELECT IDENT FROM TABLE WHERE NAME > STARTCHAR`

Right: `SELECT IDENT FROM TABLE WHERE NLSSORT(NAME) > NLSSORT(STARTCHAR)`

With a modern database manager, ORDER BY logic may remain unchanged as long as the language environment variables are set correctly. The text will be put into the correct sequence for external display.

## Visual Representation of Text

The design of an international package must take into account the way that text in different scripts will be visually represented on screens and reports. This section discusses glyphs, fonts and layouts.

### Glyphs

A "glyph" is a visual representation of a character. In the Latin alphabet, the first letter of the first word in a sentence, and the first letter of a proper name, is represented by a capital form of the letter. In Arabic, letters take different forms if they are written on their own (isolated) or take an initial, medial or terminal position in the word. In some languages, two letters may be represented by a single glyph when they appear together. Thus "ß" may be used in place of "ss" in German.

The programmers do not have to worry about glyphs as long as they follow the rules for text manipulation given above. Text will be displayed using the same glyphs that were entered. The basic rule is: Never convert text to upper or lower case. Results from functions like UPPER and LOWER will be unpredictable. Keep text in the format it was entered. Use functions like NLSSORT to compare text.

### Fonts

Different fonts present different visual appearances. Latin text may be written in Arial, Bauhaus, Brush Script and so on. The choice of font has no effect on the meaning or dictionary sequence of the text.

Although using different fonts can enhance the appearance of forms and reports, you have to be careful. There is no such thing as Times New Roman Arabic. Never hard-code a font name. Instead, refer to a data element holding the font name, where the data element value can be set differently for software builds in different national languages. The data element name should indicate the function of the font, rather than the font itself. Thus:

Wrong	<code>out_font = 'times new roman'</code>
Wrong	<code>out_font = fnt_times_new_roman</code>
Right	<code>out_font = fnt_heading1</code>

Never use a font to imply meaning. Your display must be just as meaningful (although perhaps not as pretty) if all the text is displayed in the same font.

### Form and Report Layout

Although the East Asian scripts are compact, the characters are complex and must be displayed using a large font. Forms must not contain too many lines, or the East Asian users will be forced to scroll to fill in the form.

Icons should be purely symbolic. An icon that contains letters or words will be jarring to a user accustomed to a different writing system.

## Text Management

All static text that will be displayed on the GUI or on reports must be held in external resource files or tables in the database. Field labels, messages and help files are examples of static text. The developers must isolate static text from the code.

Messages generated by system software (e.g. database manager exceptions) should never be returned direct to the user. They should always be explicitly handled by code that translates them into messages from the external table - good practice anyway but essential for non-Latin versions of the package.

Before the package can be made available in different national languages, you have to set up procedures to maintain text resource files for each language, synchronized with the software for each release, and to combine software with the different resource files during release build. It is best to design these procedures so that people in the target country can undertake translation and insertion of the local text into the software.

## Glossary

Many text elements are short. The translator will often be unable to determine the correct translation unless context is supplied. Many words can only be translated correctly when the meaning of the package jargon is known. A glossary, useful in itself, is essential in ensuring that text is translated into consistent and meaningful local language equivalents. Employ a business expert fluent in the national language to prepare the glossary for each local implementation.

## Semantics

Translation will be much easier if all national language text is clear, unambiguous and consistent. If the original text is poor, the translation is likely to be incorrect. All text should be formal, explicit, consistent and avoid the use of slang or bias. All text must conform to the glossary.

Examples of semantic considerations before translation starts:

- A generic message like "Illegal value" is not explicit. The translation could insult the end user. Validation routines should not share generic messages like this. They should each use unique messages, such as "Product quantity is not numeric".
- A data element should not be labeled "customer id", "cust identifier", "customer" on different forms. The forms must always use the same label as defined in the glossary.

It is worth investigating commercial translation software that:

- Scans the original text checking that it conforms to a well-defined set of standard sentence structures, reporting unrecognized structures. The text must be cleaned up to eliminate all non-standard structures.
- After text clean up has been performed, automatically translates the text, using national language translations defined in the local glossary for words and phrases that



support for international character sets. Versions of the technology are supplied in most character sets, including Japanese, Traditional and Modern Chinese, Korean, Arabic, Spanish and so on. Generally, the built-in panels conform to local language, including icons, text/number direction, help files and so on. A "well behaved" application, using Unicode and using the development tool's variables to determine behavior where needed, should automatically pick up the local language.

However, the developers will have to research what is meant by "well behaved", and develop a checklist for using the chosen development tools. Expect that when the first attempt is made to translate the package into Japanese, hidden linkages to Latin character sets will appear. The guidelines will have to be refined and the package will have to be overhauled. Up-front research will not eliminate all problems, but it will greatly reduce the number of problems.

### ***ASCII Interfaces***

Some otherwise good packages have not been internationalized. You may have to supply interfaces to these packages. This is not worth worrying about until the problem arises. At that time, you will have to build table-driven mapping logic, with facilities for the end user to enter the ASCII values expected by the interfacing package and the corresponding Unicode values to be used in your package.

### ***Dates***

Many countries use the Gregorian calendar, including China, Japan and Korea. Moslem countries generally use the Hegira calendar, a lunar system with dates starting from July 16, 622 AD, when Mohammed fled from Mecca to Medina. All code must be neutral to the calendar system.

The package can use the concepts of year, month, day and day-of-week, which are common to the Gregorian and Hegira systems. There are twelve months in the year in both systems, no more than 31 days in each month and seven days in each week.

However, the software must always define dates as type date, and must always use the built-in date manipulation functions in the development technology. It must never make assumptions about the number of days in the month, or which days of the week are the weekends. This logic has to be table-driven, with the ability to pick up calendar rules in the build for a given national language.

### ***Number Display***

In some countries, the normal display convention for large numbers is 123,456,789.00. In other countries, the convention is 123.456.789,00. The delimiters for large numbers and for the decimal position must be held in external variables that can be set in the build for a given national language.

Although Arabic characters are displayed from right to left, numbers are displayed from left to right. Forms must distinguish between character and number fields to ensure that the fields are filled in the correct direction as the end user types into them.

### ***User-Developed Reports***

Some database managers, such as Oracle, support Unicode table and column names. When delivering a version of the package in a given national language, it is good practice to convert the reporting database definition and the report programs to use table and column names in the target national language. This should be a simple, mechanical job.

## **Conclusions**

It is not difficult to build software that can support multiple national languages. The essential rules are:

1. Store all text in UTF-8 format
2. Use Unicode functions for all text manipulation: stringing, character counting, comparison and sequencing
3. Never convert text to upper or lower case - keep it the way it is entered
4. Use fonts, but use them with caution
5. Space out report and panel layouts to allow for large East Asian characters
6. Create and follow detailed checklists for use of each programming tool or language used in package construction
7. Use built-in language functions to manipulate dates - do not roll your own
8. Set up external resources files to hold all static text, with procedures to combine the text with the software at build time
9. The external resources files should also show the calendar to be used and the format of long numbers
10. Maintain a glossary of all jargon words for use in translation
11. When the time comes to deliver the package in a second national language, set up a formal process for text translation and software builds in each target language.

These rules add little or nothing to development costs if they are followed from the start. But if these rules are not followed, adding support for new national languages can be a major headache.