Internationalization and Localization Using Microsoft .NET

NICHOLAS SYMMONDS

Apress™

Internationalization and Localization Using Microsoft .NET

Copyright ©2002 by Nicholas Symmonds

ISBN (pbk): 1-59059-002-3

Printed and bound in the United States of America 12345678910

Technical Reviewer: Brian Jones

Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson

Managing and Production Editor: Grace Wong

Project Manager: Tracy Brown Collins

Copy Editor: Anne Friedman

Compositor: Impressions Book and Journal Services, Inc.

Indexer: Rebecca Plunkett

Cover Designer: Tom Debolski

Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010

and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA 94710.

Phone 510-549-5938, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 2

Aspects of Localization

THIS CHAPTER INTRODUCES some of the more common aspects of designing a multilingual program. You need to be aware of how dates, time, numbers, and calendars are affected by region. Of course, the one part of the program that may be as big as the program itself is the Help system. These elements are quite often left out of an internationalization project and if so could cause quite a bit of animosity to your program. The code for the programs in this chapter, as in the whole book, can be downloaded from the Apress web site at `http://www.apress.com`. The final section of this chapter introduces you to the Unicode standard, what it is, and how you may already be using it.

The most important part of any program is probably the design of the GUI itself, which I discuss next.

GUI Design for Mulitinational Programs

I would like to talk about basic GUI (Graphic User Interface) design strategy. There are tons of books available on how to design GUIs that contain rules for what you should and should not do. What I want to touch on here is GUI design specifically in relation to multilanguage programming.

I am sure most of you have experience coming up with screens, at the request of the marketing department, for those incredible programs you are developing.

Many demo screens, however, also tend to be the basis for the finished product. How many times have you shown a demo screen to your boss to explore a concept and then gone back and built the code around these same screens?

What is wrong with this approach? Most likely you have laid out the screens to be just the right size for the English words and phrases you use. The design needed for localization gets left out.

Keep in mind the lengths of the strings you use. It is common in GUI design to use short sentences or single words to label fields. Translated text can be considerably longer than the original English text. It is actually an inverse relation depending on how long the original text string is. Table 2-1 shows how much the string length will grow when translated.

Table 2-1. Buffer Size Growth Based on Original String Length

ENGLISH	OTHER
1 to 5	100%
6 to 20	70%
20 to 50	30%
> 50	15%

As you can see, the shorter the string length the more space in relation to the original string you need.

As you research some of the languages into which you will convert your program, you will find that some languages need entire phrases to literally translate one English word. The reverse is also true. You need to plan a little in the design of your GUI to allow for this kind of situation.

Here are some examples of single English words translated into German. While these words may not be typical of words you would use in a program, they give you an idea of the difference between languages.

Table 2-2. Some English Words and German Translations

ENGLISH	GERMAN	BUFFER GROWTH
Watch	Bewachung	80%
Obsolete	nicht mehr gebrauchlich	287%
Textiles	bekleidungsindustrie	250%

What about phone numbers and addresses? The ISO standard for the length of a phone number is 15 digits. Be sure to allow some extra room for things such as Private Branch Exchange (PBX) codes and country codes. Appendix A lists phone country codes.

Do not assume that the dash is the only number separator in a phone number. You need to allow spaces, dashes, commas, and periods.

An address in the United States includes some information that makes no sense elsewhere. Take for instance a state. This means nothing in Taiwan. It just adds a level of confusion. Be flexible with your address format and allow enough fields to locate an address anywhere.

In the United States a ZIP code is a 5+4 digit number with the extra 4 digits being optional. Be careful not to validate a ZIP code based just on this pattern.

Quite a few other countries use letters in their postal code and they may also be of differing lengths.

Here is the current address for Oxford University in England. Notice the U.K. equivalent of the ZIP code. If you do not allow for alpha characters your correspondence might never get there.

University of Oxford

University Offices

Wellington Square

Oxford. OX1 2JD. UK.

Message Boxes, Dialog Boxes, Maps, and Menus

Message boxes are used extensively in many programs and quite often the text they display is long. Message boxes also resize depending on the amount of text shown. A small message box in English could be quite large in German.

Dialog boxes may also grow, especially some of the common dialog boxes. If you can, try to make your dialog boxes large enough to prevent resizing controls. Plan for a text box that can wrap the text to another line. You are better off if you can avoid having to resize the dialog box.

Dialog boxes as well as forms are much easier to understand when they are not cluttered. Spread your fields out among logically constructed screens. You may find that a screen with many fields needs extra space to allow for text expansion when translated. If you do not leave extra space you may find that your translated text wraps and makes a messy screen. If you have space limitations in your text fields make a comment in your resource file noting this fact. Let the translator find the best word or phrase that fits.

Try to make sure that phrases are not split between labels or text fields. Quite often a sentence or phrase in English swaps words around when translated. If you have one part of a phrase separate from another part in the resource file, the translator will not be able to make the correct translated phrase out of the two separate pieces. For example, German sentences often have verbs at the ends of sentences, while English and French place them in the middle.

I once took over the task of localizing a program that used a series of dialog boxes of the same size. The author ran his program by placing these dialog boxes on top of each other in a modal fashion thus hiding the screen behind it. When I localized the program some of the new strings were much longer then the original English versions and the resize control allowed for this by resizing the dialog boxes. It ended up that he could no longer hide some dialog boxes behind others because they peeked through at the sides. His attempt to hide what he was doing failed once his program was translated into another language. This was poor design indeed.

The dialog boxes in question were actually different executables. The programmer was trying to simulate multithreading in VB. I disagreed with this approach, but I was not the one making the decisions.

A menu system is one where you can really get into trouble with localization. The topmost menu items in a menu list are always meant to be displayed on the screen. If you have quite a few menu items you probably have tried to make them all fit on just one line. If you have so many choices that you needed to "make them all fit" then you need to rethink your design. The menu will most probably grow quite a bit in size when localized and you will end up wrapping your menu. This is something you definitely need to take into account.

This century has seen boundary lines on maps redrawn countless times. We have seen new countries spring up and several countries combine into one. Even today many parts of the world are in flux and border disputes abound. If you need to display a map make sure you have the latest version for that region. You do not want your program to offend anyone who may take exception to the map you show.

Fonts and Keyboards

These days Windows has a large number of fonts natively available depending on the language version of Windows you have.

You may find that some of your characters are coming out with question marks and other characters that are not what you expect. If this is the case you most likely need a new font for your program.

If you find that you are translating to languages that are not supported by the built-in fonts you may need to include them with your program. Consider localizing the fonts you need in a resource file. You can then load the font you need at runtime without cluttering up the destination PC.

Keyboard layouts change according to locale. In some countries certain character do not appear on the keyboard at all. In such cases there are shortcut key combinations that are used to get the right character. If you need to set up shortcut keys, remember to use only keys that you are sure are on the keyboard at that locale. If you want to be independent of locale then use the function keys to do this.

To summarize; here are some pointers to keep in mind when designing user screens.

- Do not split phrases between label or text controls on your screen.
- Don’t try to jam all your fields on one screen. Nothing is worse than a busy screen in English that when translated to another language has all kinds of word wrapping.
- Leave room for word expansion in your text fields.
- Truncate strings to the maximum length of a text field. This prevents unwanted word wrapping.
- Do not concatenate translated words or phrases to make sentences. Word order will invariably trip you up.
- Use proper English wherever possible. Slang translates poorly.
- Be careful of abbreviations or industry specific terms. Build up a glossary as you go along.
- Do not depend too much on the size of message boxes. They change in relation to the number of characters displayed.
- Make your dialog box large enough to handle translated text. Resizing a dialog box could lead to unexpected results.
- If you have quite a few first-level menu choices be prepared for them to wrap after being translated.
- Consider keeping nonstandard fonts in a resource file.
- Try to keep pictures of international maps to a minimum. Map divisions can be a hot point with quite a few people.

Formatting International Time

Remember when mom taught you to tell time on an analog clock? Pretty confusing when you consider that it can be 11 o’clock twice a day. Of course when you are 5 or 6-years-old you say “in the morning” or “in the afternoon.” Only later did you learn the AM/PM part.

NOTE *AM and PM stand for ante meridian/post meridian.*

That was okay for us in the United States. What about overseas? Most nations have standardized on military time. Most of us here in the United States only know it through John Wayne war movies where he asked people to synchronize watches at 0600 hours. Go to Europe and 9 PM is most always 21:00 when written. When you think of this in terms of programming, military time is definitely easier to work with (sort, add, subtract).

Try to make an algorithm to take the difference between 10 AM and 3:45 PM. It takes a little doing in analog time but military time is trivial.

Okay, I know what you are thinking. What happens at midnight? Well both 00:00:00 and 24:00:00 mean the same thing. However to remove ambiguity you should refer to midnight as 00:00:00. Digital clocks do not display 24:00:00.

In general the world standard for time is hh:mm:ss. Where hh is the number of complete hours that have passed since midnight (00-24), mm is the number of complete minutes that have passed since the start of the hour (00-59), and ss is the number of complete seconds since the start of the minute (00-60). If the hour value is 24, then the minute and second values must be zero.

Formatting Dates

Want a date? How about “3/4/05”? What date is this? Is it March 4, 2005 or March 4, 1905 or April 3, 2005 or April 3, 1905. Any of these interpretations is feasible depending on your location and your age.

Time is basic and you can pretty much tell what someone means when it is displayed. As you can see, dates are a different story.

You might see dates in the following formats. 8/7/99, 7/8/99, 99/7/8, 8.7.1999, 07-OCT-1999, 7-October-1999. And there are quite a few more. It can be quite confusing.

The international date standard notation is YYYY-MM-DD. This based on the Gregorian calendar where YYYY is the year. MM is the month between 01 and 12. DD is the day between 01 and 31.

The ISO has passed a language-independent international time and date standard called the International Standard ISO 8601. Aside from solving confusion over what date notation to use, the advantages of this standard are many.

- The standard is easily readable and writeable by software (no 'JAN', 'FEB', . . . table necessary.)
- It is comparable and sortable with a trivial string comparison.
- Provides consistency with the common 24h time notation system.
- Strings containing a date followed by a time are easily comparable and sortable.
- The notation is short and has constant length, which makes both keyboard data entry and table layout easier.
- This date notation is already used in much of the world.

ISO date and time standards are very helpful to both the programmer and to the end user. Why the programmer? How many times have you had to add and subtract time or dates based on a 12-hour clock? Perhaps you have tried to find the day of the year and the number of days left in the year for a scheduling program you are writing. The algorithm for using a 12-hour clock and day and month names is very difficult. The ISO standard formats dates in the slowest moving time to the fastest. This makes date and time very easy to sort and calculations very easy to compute.

What about the end user? Quite a few countries, such as Japan, Korea, Hungary, Sweden, Finland, Denmark, and others, as well as people in the United States, are already used to at least the "month, day" order. This format is already used in much of the world. The end user also benefits from easier and more constant keyboard entry. Entering in May 24 (04-24) is the same as entering in September 24 (09-24).

Both time and dates are stored in different formats programmatically. Whatever the format, you should use some kind of formatting command to display the time based on a setting the user chooses or based on the regional settings of your computer. Both Visual Basic and Visual Studio .NET have such format commands.

Formatting Dates in Visual Basic 6

Visual Basic had some basic date formatting parameters that converted a date value to text based on the regional settings of your computer. An example of this is:

```
format ( now(), "General Date" )
```

This returns a string representation of the date and time according to your system settings. Other date formats that are displayed according to system settings are:

- "Long Date"
- "Medium Date"
- "Short Date"
- "Long Time"

Formatting Dates the .NET Way

The .NET way of doing this is somewhat different. .NET does not need a separate function to handle transformation of basic data types.

Now for a little review on .NET architecture. The most basic lesson of .NET is this: Everything inherits from the object class. . .*everything*. This means that all basic data types you are familiar with are actually objects. This includes integers, strings, longs, and dates.

As you study the basic data types in .NET you will find there are two kinds: value and reference types. The short explanation is that they can be treated the same ways. If you declare an integer and use it only for simple math operations it stays a value type. If you want a little more out of it such as determining what type it is then through the magic of "boxing" it becomes a reference type. It is now an object. I encourage you to review the documentation on boxing and play with boxing until you understand it. Knowing when a type is boxed and unboxed can make a difference in how you program a particular algorithm.

In VB a date is essentially a double. This stems from the fact that VB 6 is COM-based, and a date in the COM world is an OLE automation date, which is a double. All types in .NET inherit from the base object class. Because of this, the date type is also an object. All well-written objects (.NET has only well-written objects) have a certain amount of what I call "programmed instinct." They know what they are and what they are capable of doing. Most good objects can also transform their data to another form if appropriate.

This is all true for the date type (object). If you want to print out a date in one of several formats you would use the following piece of code.

C# Example:

```
DateTime MyDate = new DateTime(2001, 8, 2);
MyString = MyDate.ToString("F");
```

VB .NET Example:

```
Dim MyDate = new DateTime(2001, 8, 2)
MyString = MyDate.ToString("F")
```

The result of this code would be "8/2/2001" if the current culture was U.S. English.

The DateTime structure has seven overloaded constructors. You can initialize it with just about any kind of date or time you can think of. As you can see, the DateTime object can return a string according to the current culture settings. A partial listing of output formats with default patterns are:

• "d"	M/D/YYYY
• "D"	dddd, MMMM dd, yyyy
• "s"	yyyy-MM-dd HH:mm:ss

The last one conforms to the ISO standard 8601. There are quite a few others, and I encourage you to visit the VS .NET Help files to familiarize yourself with them.

Whatever you do in regard to displaying dates and times, make sure you are consistent throughout your program.

The Calendar

There are several calendars still in use around the world. The most popular is the Gregorian calendar. The Gregorian calendar was devised as a way to fix the problems with the Julian calendar. These problems had to do with the way Easter was calculated and the length of the tropical year. The Julian calendar lost one day every 128 years. Although the Julian calendar was dropped by most of the world in the 1500s, it is still used today by the Russian Orthodox Church and other orthodox churches.

The main calendars in use today are:

- Hebrew
- Chinese
- Japanese
- Julian

- Gregorian
- Islamic
- Balinese
- Baha'i
- Ethiopian

While the details of each calendar are out of the scope of this book I will say that most of the time the Gregorian calendar is the predominant one. Visual Studio .NET does allow date calculations in other calendars. If you find that you are making a date-centric application such as an HR program, it would behoove you to make use of these functions.

The System.Globalization namespace in VS .NET has the following calendar implementations.

- GregorianCalendar class
- HebrewCalendar class
- HijriCalndar class
- JapaneseCalendar class
- JulianCalendar class
- KoreanCalendar class
- TaiwanCalendar class
- ThaiBuddhistCalendar class

Each of these classes allows manipulation of dates within the particular calendar you are working with. This is not only way cool but allows easy implementation of different calendar types within your program. The good folks at Microsoft have done all the complicated calculations for you.

Numbers and Currency

This can be quite a confusing subject. In VB 6 formatting a number according to local depends on your computer's setting. You cannot define programmatically that a group separator is a comma or a period. The same goes for the decimal separator. If you used the piece of code

```
X=123,456.78
S=Format  ( x, "###,###.###" )
```

you get the string 123,456.78 in the United States, but if your computer is set for Germany you get 123.456,78. The comma and period are swapped.

TIP *Suppose you use a text box for real number input below 1000. The standard method is to catch each keystroke and verify that it is a digit or a decimal point. Wrong! This works in the United States, but it will not let someone in England input any number other than an integer. You must also allow a comma.*

VS .NET has string-formatting commands built into the numerical data types. Just like the date and time issue, there is no separate function needed in VS .NET to convert a number to a string. Again this is done using the ToString method associated with these objects. By the way, the ToString() method is Unicode-aware.

Let's look at some code to see how the ToString() function works. The first sample shows this function in VB .NET:

```
Dim ThisInt as integer = 12345
Dim MyString as string = ThisInt.ToString( "c" )
MyString = ThisInt.ToString( "d7" )
MyString = ThisInt.ToString( "g" )
```

Here is the C# example:

```
int MyInt = 12345;
String MyString = MyInt.ToString( "c" );
MyString = MyInt.ToString( "d7" );
MyString = MyInt.ToString( "g" );
```

The first MyString would be “$12,345.00”.
The second MyString would be “0012345”.
The third MyString would be “12345”.

As you can see, the format specifier allows you to represent the number in any of several ways. How do you make this internationally aware? The answer is in the System.Globalization.CultureInfo namespace. You can initialize the constructor with a code for a country, and the MyInt.ToString() member swaps the comma and decimal point if appropriate.

The ToString() conversion function for all basic data types is culture-aware. However, if you use a format specifier without a corresponding argument for the culture, the resulting string is formatted according to the culture that your system is set to in the regional settings of the control panel. If you are making a program that will be able to swap languages at runtime then you need to add this extra argument to all your ToString() commands. The following code takes the previous number and currency example and makes it internationally aware. It also allows you to change culture via code, which essentially gives you runtime control over changing languages.

This example is shown here in VB, but it is available for download in C# if you wish.

VB .NET Example:

```
Imports System.Globalization
. . .
        Dim mystring As String
        Dim MyCulture As CultureInfo
        Dim thisdate As DateTime = #8/2/2001#
        Dim ThisInt As Integer = 12345

        'The current culture of the computer
        MyCulture = CultureInfo.CurrentCulture
        mystring = thisdate.ToString("d", MyCulture)
        mystring = ThisInt.ToString("c", MyCulture)
        mystring = ThisInt.ToString("d7", MyCulture)
        mystring = ThisInt.ToString("g", MyCulture)

        'The German culture
        MyCulture = New CultureInfo("de-DE")
        mystring = thisdate.ToString("d", MyCulture)
        mystring = ThisInt.ToString("c", MyCulture)
        mystring = ThisInt.ToString("d7", MyCulture)
        mystring = ThisInt.ToString("g", MyCulture)
        'The US culture
        MyCulture = New CultureInfo("en-US")
        mystring = thisdate.ToString("d", MyCulture)
        mystring = ThisInt.ToString("c", MyCulture)
        mystring = ThisInt.ToString("d7", MyCulture)
        mystring = ThisInt.ToString("g", MyCulture)

. . .
```

Let's describe a little about what is going on here.

The first thing I do is import the System.Globalization namespace. This gives me access to the classes under this namespace without having to resort to using the full name. I could have made a reference to another assembly that imported this namespace and achieved the same thing. After this I set up a variable that will hold the current culture as well as some data variables to work with.

Let's look at this first block of code.

```
        'The current culture of the computer
        MyCulture = CultureInfo.CurrentCulture
        mystring = thisdate.ToString("d", MyCulture)
        mystring = ThisInt.ToString("c", MyCulture)
        mystring = ThisInt.ToString("d7", MyCulture)
        mystring = ThisInt.ToString("g", MyCulture)
```

The current culture is set to U.S. English. For the first block of code the variable mystring will have the following values:

1. "8/2/2001"

2. "$12,345.00"

3. "0012345"

4. "12345"

The next block of code changes the MyCulture object to be German.

```
        'The German culture
        MyCulture = New CultureInfo("de-DE")
        mystring = thisdate.ToString("d", MyCulture)
        mystring = ThisInt.ToString("c", MyCulture)
        mystring = ThisInt.ToString("d7", MyCulture)
        mystring = ThisInt.ToString("g", MyCulture)
```

For this block of code the variable mystring has the following values:

1. “02.08.2001”

2. “12.345,00”

3. “0012345”

4. “12345”

Notice that my code is the same except for changing the culture. The date and currency format changed according to the culture I set.

Notice something interesting about the currency? As of the time I am writing this book the German currency is German Marks. However .NET is anticipating the changeover in 2002 from Marks to Euros.

So what do I do if I want to express money in German Marks? Well as it so happens there is a way to do this by making your own number format class and passing it to the CultureInfo class. Consider this piece of code.

```
'Here is how to represent the old German currency format
Dim OldGermanFormat As New NumberFormatInfo()
OldGermanFormat.CurrencySymbol = " DM"
OldGermanFormat.CurrencyDecimalSeparator = ","
OldGermanFormat.CurrencyGroupSeparator = "."
OldGermanFormat.CurrencyPositivePattern = 1
OldGermanFormat.CurrencyNegativePattern = 1

'The current German culture
MyCulture = New CultureInfo("de-DE")
MyCulture.NumberFormat = OldGermanFormat

mystring = ThisInt.ToString("c", MyCulture)
```

The resulting value of mystring is “12.345,00 DM”. Just what I wanted.
Let’s look at what I did here:

1. I set up an object as a NumberFormatInfo class.

2. I set the decimal separator and group separator to be the same as is used in Germany.

3. I made the CurrencySymbol something that represents the old German Mark. The default for this is the “$.”

4. I set the positive and negative pattern for the currency so that the number precedes the symbol.

5. I set the current culture to German. (More than numbers are involved here in a language.)

6. I set the current culture to German. (More than numbers are involved here in a language.)

7. I set the internal number format of the current culture to be OldGermanFormat.

The flexibility included here allows you to pretty much do what you want. Microsoft included just about every known modern culture in the world, but even they can't predict the instability in different regions. By the time this book is published there may be a new country or two to deal with.

How Sort Order Is Affected by Language

There are two basic types of sort orders for strings. The first is ASCII sort order. This is where the strings are sorted according to their letter placement in the ASCII table. Letters are placed in the ASCII table capital letters first. In this case, words that begin with A, b, C would be sorted as A, C, b.

The International sort order is defined as being case-insensitive. So the true sort in the above example would be A, b, C.

There are quite a few other types of sort orders within the international arena. These are all language-based. Some of these are the Czech, Swedish, Danish, Polish, Spanish, French, and so on. Most of these sort orders have different rules for the diacritical marks. Some define a character with a diacritical to come before the same character without, and some are the reverse. Also because some European languages have more letters than English, there are cases where what would seem normal sort order to someone in the United States is totally different to someone in Russia.

Suppose you had the following words sorted in normal International sort order:

- Victory
- Wake
- Woman
- Yak

If you sort them in Finnish sort order they would be arranged like so:

- Wake
- Victory
- Woman
- Yak

In Finland the V is considered the same level as the W. This could really play havoc with your database indexes. Watch out for this.

You can get the sort key string that defines sort order from .NET. It is under the System.Globalization namespace. Look in the SortKey class under OriginalString. The KeyData and OriginalString members can both be overridden to make your own sort order.

Creating International Help Files

Back in the days of DOS, most programmers did not write Help files for their programs. Not much of an issue here. When Windows came along all of a sudden we got context-sensitive Help. Pushing the F1 key while on a field, screen, or even a word would bring up Help that was what you wanted. No more of this pulling up the Help file as a whole and trying to find a topic that addressed your needs.

NOTE *Full coverage of Help files and how to create them is beyond the scope of this book. Instead I wish to convey some more philosophical aspects of Help file creation.*

Unless your program is the most intuitive in the world, you need a comprehensive Help system. Believe me when I say that the Help file can make or break a good program.

It can also greatly reduce those pesky tech support calls. (But then what is the point because no matter how good the Help is no one ever reads the manual anyway. . .but I digress.)

Make sure to use the same translator, or at least the same translating project manager, for your program strings as well as your Help files. Many English words and phrases can have several meanings in different languages. If you translate a sentence from English to Chinese in your program, make sure that the same sentence in your Help file is translated the same way. If not you run the risk of the Help file adding confusion instead of clarity.

By the same token, have the program and the Help file translated at the same time. Always keep them current with each other. The thought process necessary in converting your text files should be the same one used in converting your Help files. If you decide to translate the Help files six months after the strings then the translator will have probably forgotten some of the nuances involved at the time he or she translated your strings.

As a programmer you probably should not be doing your own Help file. You will instead need to work closely with a tech writer to accomplish this. To summarize, here are a few hints to follow as you work with the designer of your Help system.

- Keep the explanations as free of jargon as you can.
- Make sure that any screen shots can be easily replaced. It is no good translating the Help file while showing screen shots in English.
- Be sure to use the same translating project manager for your program strings as well as your Help files.
- Have the program and the Help file translated at the same time.

Introducing Unicode and Character Sets

What is Unicode? Perhaps you think it is one of those persistent buzzwords that just won't go away. Believe me when I say it is not a buzzword. As far as multilingual computing goes, Unicode is the most important thing to ever come along in the computer business. Unicode is one of those things in the computer industry that is slowly being adopted with hardly any fanfare. In fact, for quite a few programmers, Unicode is largely unseen and unnoticed.

So what is Unicode? Unicode is a way to provide a unique number that identifies every single character in every human language. There is even room left over for Klingon!

Let's back up a step. You have certainly worked with the ASCII table. Consider the following piece of VB code:

```
Dim letter As String
Dim number As Integer

letter = Chr(65)
number = Asc("A")
```

This code converts an ASCII number to its character representation and back again. In ASCII the capital letter A is 65. If you have ever intercepted a key press event from one of the VB controls you have had to use the ASCII conversion routines to see what letter was pressed.

While the ASCII table has 256 character representations, most of us only program with the lower 128. This is mainly because it is enough to write most anything in the English language. If you look at the ASCII table you see that the upper 128 characters are a collection of some foreign characters, punctuation, lines, and blocks.

NOTE *It was very common in the DOS days to draw menus and graphics on the screen using the upper 128 characters of the ASCII table. In fact quite a few programmers, myself included, could recite most of the ASCII table by heart.*

Code Page Usage

In the days before Unicode Version 1 was fully adopted, programmers used code pages to display characters from different languages. Code pages are still supported in Windows but are only really used for older programs. A code page is a different interpretation of the ASCII character set. Code pages keep the same lower 128 characters intact (mostly) but the upper 128 characters are tailored to a particular language. There are many Windows code pages as well as DOS code pages. This means that the ASCII character for #180 is different for almost every code page. In fact the Cyrillic code page for DOS is different than the Cyrillic code page for Windows. They have all the same characters but the ASCII number is different for both. This was quite a problem for the multilingual programmer always having to keep track of what code page you might be working from. Imagine trying to send a text file that was rendered using a certain code page to someone. Chaos could easily ensue if the person you sent it to was not up on code pages.

I once had to send out English text to be translated into Cyrillic to be used on an embedded system. There were constant phone calls and emails about which code page was being used and how to represent it. The embedded target system used a DOS Cyrillic code page 866 and I got the translations back in a Word doc that used the Windows Cyrillic code page 1251. Do this just once and you understand the need for Unicode.

In Windows 9*x*/2000, code pages could be switched on the fly without having to change language. In DOS you had to change the code page with some DOS commands. Needless to say, using code pages was not the most elegant way of enabling different character sets to be displayed on your screen.

I could go on and make this book quite a bit heavier with code page information. However the preferred method is definitely to use Unicode. Because .NET is totally new and Unicode-based I will not go into any more depth on code pages.

Relating Double Byte Character Sets to Unicode

What about Eastern languages where Chinese for instance has over 5000 characters? A different scheme was invented for this based on the concept of code pages that contain 256 code points. The result is called the Double-Byte Character Set (DBCS).

In DBCS, a pair of code points (a double-byte) represents each character. The first byte of a double-byte set was not considered valid unless it was followed by a second byte defined in the DBCS set. DBCS required code that would treat these pairs of code points as one character. This still disallowed the combination of two languages, for example, Japanese and Chinese, in the same data stream because the same double-byte code points represent different characters depending on the code page. DBCS was used for some time but is now going out of style.

Along comes our saving grace Unicode. Unicode is based on the ASCII table for compatibility but greatly extends it. Instead of being one byte in length Unicode represents characters with 2 bytes. This 16-bit encoding scheme means that codes are available for 64k characters. While this number is sufficient for coding the characters used in the major languages of the world, the Unicode Standard provides the UTF-16 extension mechanism (called surrogates in the Unicode Standard), which allows for the encoding of as many as 1 million additional characters. This is sufficient for all known character encoding requirements, including full representation of all historic scripts of the world. This brings order to the chaotic world of character representation.

The first 128 characters of Unicode are the normal Latin ASCII character set. These characters go from 0000 to 007F hex. In Unicode the word “dog” would be represented by 0064006F0067. This plays havoc with C code because in C a string is terminated with a NULL character, which is 00. As you can see Unicode is not compatible with normal C strings.

To reiterate, Unicode assigns a unique letter for every character without regard to:

- Language
- Computing platform
- Program

This is quite an accomplishment considering all the disparate computing systems in the world.

Programming with Unicode

NOTE *This is just a scant introduction to Unicode. Many pounds of books have been written about Unicode. I suggest you get the Unicode 3.0 book put out by the Unicode consortium. It is a valuable reference.*

If you are a VB programmer you have been using Unicode since Version 5. All Visual Basic strings are represented internally in Unicode. VB has been ready, willing, and able to help

in localization for years.. How can you tell that your string is represented in Unicode? Try the following VB example.

```
x = Len("Unicode")
x = LenB("Unicode")
```

The first line sets *x* to 7, the number of letters in the word Unicode. The second line sets *x* to 14. This is the number of bytes needed to store the word Unicode. In ASCII 7 bytes would be enough. For you C lovers, an 8th byte would be needed to store the null terminator.

Visual Basic always did the Unicode to ANSI translation for you transparently. Windows NT and higher operating systems from Microsoft are fully Unicode compliant. Visual Studio .NET is fully Unicode compliant. You now have a great basis for writing programs in .NET that will work anywhere in the world.

So how can you see the power of Unicode? There is a great Unicode editor called `UniEdit`. This program was developed in conjunction with Duke University. There is currently a free trial version of UniEdit available at `http://www.humancomp.org/uniintro.htm`. The cost for buying it is minimal and its usefulness is infinite.

NOTE *I have occasional need to reference the official Unicode book. I often find it fascinating to flip though and look at other writing systems. A 2-minute lookup may take me $\frac{1}{2}$ hour. I am the same way with the dictionary.*

How do you know that .NET is Unicode just by glancing at the documentation? Look at the documentation concerning data types. A Char is now two bytes. It is big enough to hold a UTF-16 Unicode character. Traditionally it had always been one byte.

NOTE *I can't tell you how many programs I have written (embedded and DOS) that counted on the fact that a char was one byte. So many algorithms that involved counting were based on this fact.*

Summary

This chapter dealt with some of the more prevalent concepts surrounding localization. I talked about what is necessary to properly format and display your information to the user. Data presentation is arguably the most important aspect of a program.

I ended up this chapter with a short discussion of Unicode and how prevalent it is in both programming languages and in the operating system itself.

Some things to remember are:

- Make sure your text boxes are able to handle translated strings that can be anywhere from 20 to 100 percent of the original English size.
- Make sure that numeric input allows the interchange of a comma with a period as demarcation identifiers.
- Allow for growth in the size of dialog boxes, message boxes, and menus.
- .NET is Unicode aware. Learn what Unicode is and use it to your advantage.
- Be aware of different time, date, and numeric formats for different cultures.
- Do not depend on the U.S.'s standard sort order. Some cultures sort strings in a different order. Make sure your program takes this into account.
- Do not forget the Help files. Translating them in synch with the program strings can avoid confusion between different translations of the same phrases.

There are many of you who will spend some time in between programming languages as you slowly migrate to .NET. In Chapter 3 I cover how to use multiple resource files in VB 6. I also show you how to manage these resource files in a manner similar to .NET.