If you are planning to distribute your Visual Basic application to an international market, you can reduce the amount of time and code necessary to make your application as functional in its foreign market as it is in its domestic market. This chapter introduces key concepts and definitions for developing international applications with Visual Basic, presents a localization model, and emphasizes the advantages of designing software for an international market.

This chapter also discusses guidelines for writing Visual Basic code that results in a flexible, portable, and truly international application. A section is devoted to writing Visual Basic code that handles the specific aspects of the double-byte character set (DBCS) used on East Asian versions of Windows.

Note If you are using the Control Creation Edition of Visual Basic, some of the material covered in this document may not be applicable. With the full editions of Visual Basic you have the ability to create applications, ActiveX documents, and other types of components. Although some of the terminology may relate to application specific objects such as forms, in most cases the underlying concepts also apply to ActiveX control creation.

1

Contents

2

Sample Application: Atm.vbp

Some of the code examples in this chapter are taken from the Automated Teller Machine (Atm.vbp) sample. If you installed the sample applications, you'll find this application in the \Atm subdirectory of the Visual Basic samples directory (\Vb\Samples\Pguide).

International Software Definitions

Before you start developing international software, you should know some fundamental terms.

International Software

International software is software that is marketable worldwide. A software product is international only if it is as functional in its foreign market as it is in its domestic

market. For more information about how to localize your application, see "Designing International Software" later in this chapter.

Locale

A *locale* describes the user's environment — the local conventions, culture, and language of the user's geographical region. A locale is made up of a unique combination of a *language* and a *country*. Two examples of locales are: English/U.S. and French/Belgium.

A language might be spoken in more than one country; for instance, French is spoken in France, Belgium, Canada, and many African nations. While these countries share a common language, certain national conventions (such as currencies) vary among countries. Therefore, each country represents a unique locale. Similarly, one country might have more than one official language. Belgium has three — French, Dutch, and German. Therefore, Belgium has three distinct locales. For more information about locale-specific settings, see "General Considerations When Writing International Code" later in this chapter.

Localization

Localization is the process by which an application is adapted to a locale. It involves more than just literal, word-for-word translation of these resources — it is the meaning that must be communicated to the user. For more information about how to localize your application, see "Designing International Software" later in this chapter.

String Resources

String resources refers to all the text that appears in the application's user interface. They include, but are not limited to, menus, dialog boxes, and informational, alert, and error messages. If an application will be used in a locale other than the one in which it was developed, these resources will have to be translated, or localized.

For More Information For definitions of East Asian terminology, see "ANSI, DBCS, and Unicode: Definitions" later in this chapter. For more information about string resources and resource files, see "Using Resource Files for Localization" later in this chapter.

Designing International Software

It is a lot more efficient to design your application with localization in mind, following an approach that separates string resources from code, than to revise your finished application to make it international later in the development process.

Advantages of Designing International Software

There are four primary advantages to designing and implementing your Visual Basic application so that it is sensitive and appropriate to international conventions, foreign data, and format processing:

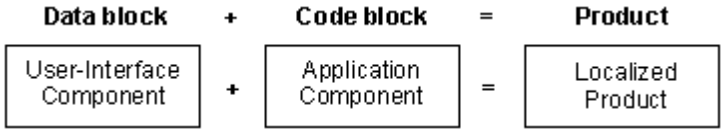
- You can launch your Visual Basic application onto the market more rapidly. No additional international development is needed once the initial version of the product is complete.
- You use resources more efficiently. Adding international support to your finished Visual Basic application may make it less stable. More development and testing resources would be required than if the same support had been implemented as part of the original development process.
- If the international version of your Visual Basic application is built from the same set of sources as the version in which you originally developed your application, only isolated modules need international attention, thus making it easier and less expensive to maintain code while including international support. See "Using Resource Files for Localization" later in this chapter.
- Developing an international version of your application becomes easy. For instance, you can develop an English-language version of your application that runs in a German operating environment without rewriting code. You only need to customize the user interface. See "Designing an International-Aware User Interface" later in this chapter.

1

Localization Model

Any application that will be localized represents two conceptual blocks: a *code block* and a *data block*. Figure 16.1 represents the data block as the "user interface component" and the code block as the "application component."

Figure 16.1 The data block and code block make up a localized product



2

The data block contains all the user-interface string resources but no code. Conversely, the code block contains only the application code that is run for all locales. This Visual Basic code handles the string resources and the locale-specific settings. "Writing International Code in Visual Basic" provides details on how to write Visual Basic code that handles locale-specific settings, such as dates, currencies, and numeric values and separators.

In theory, you can develop a localized version of your Visual Basic application by changing only the data block. The code block for all locales should be the same. Combining the data block with the code block results in a localized version of your application. The keys to successful international software design are the separation of

the code and data blocks and the ability for data to be accurately read by your Visual Basic application, regardless of the locale.

Although it may be more work for the person writing the Visual Basic application, no user-interface elements should be present in the Visual Basic code. Instead, the string resources should be placed in a separate file, from which they will be loaded at run time. This file is called a *resource file* (.res), which is a file that contains all the strings, bitmaps, and icons that are localized. For more information about resource files, see "Using Resource Files for Localization" later in the chapter.

The teams working on localizing the application should work exclusively on the resource file to develop all the different language versions of the application. This approach has the following advantages:

- **Efficiency**. Developing a new international version of the application only involves creating a new international resource file because each version has the same code block. This streamlines the creation of multiple language versions of your Visual Basic application.
- **Greater security**. Whether you decide to localize your application in-house or to use an external company, you won't need to access source code to develop international versions of your application. This approach will also lower the amount of testing needed for the international version.
- **Better localization**. By placing all string resources in one file, you ensure a more efficient localization process and reduce the chance of leaving some strings unlocalized.

3

The following table lists some factors to consider when designing your Visual Basic application.

Factor	Item
Language	Strings in the user interface (menus, dialog boxes, and error messages)
	Printed and online documentation
Locale-specific settings	Date/time formats (separators, order of day/month/year)
	Number formats (decimal and thousand separators)
	Currency formats (symbol and format)
	Sort order and string comparison

3

The first factor, language, is addressed primarily in the design phase of your Visual Basic application. See "Designing an International-Aware User Interface" for more information. The second factor, locale-specific settings, is discussed in "Writing International Code in Visual Basic" and "International Sort Order and String Comparison" later in this chapter.

Using Resource Files for Localization

—4

A resource file is a useful mechanism for separating localizable information from code in Visual Basic.

Note You can have only one resource file in your project. If you attempt to add more than one resource file, Visual Basic generates an error message.

4

Advantages of Storing Strings in Resource Files

When you are writing Visual Basic code, you can use the LoadResString, LoadResPicture, and LoadResData functions in place of references to string literals, pictures, and data. Storing such elements in a resource file offers two benefits:

- Performance and capacity are increased because strings, bitmaps, icons, and data can be loaded on demand from the resource file, instead of all being loaded at once when a form or a module is loaded.
- The resources that need to be translated are isolated in one resource file. There is no need to access the source code or recompile the application.

4

To create a resource file

1 Create a resource source file (*.RC) that contains all the string resources of your application.

1The syntax for creating the resource source file is documented in Resource.txt in the \Tools subdirectory of the main Visual Basic directory. This information is also available in the Windows Software Development Kit, as well as on the Microsoft Developer Network CD. You must associate an identifier (ID) with each resource, and then reference each ID in your code.

2 Use a resource compiler to convert the resource source file into a resource file (*.res). You can use the resource compiler (Rc.exe) shipped in the \Tools\Resource\Rc32 subdirectory of the main Visual Basic directory to convert the resource source file.

5

Note In Visual Basic, the resource whose ID is 1 is reserved for the application icon. Therefore, you cannot have a resource in your .res file with that ID number. Visual Basic generates an error message if your code attempts to load that resource ID.

5

To localize a resource file

3 Load the resource file in a resource editor. AppStudio, which is shipped with Microsoft Visual C++, can be used to edit the entries.

4 Once the file is loaded, localize the entries. Create as many language versions of the strings, bitmaps, icons, and data as you need.

6

To add a resource file to your project

5 From the **Project** menu, choose **Add File** (CTRL+D).

6 In the **Add File** dialog box, select **Resource Files (*.res)** in the **Files of type** box.

7 Select the resource file you want to add to the project, and click **Open**.

7

Visual Basic recognizes resource files by the .res file name extension. If the resource file does not have the appropriate file name extension, Visual Basic won't load it. Conversely, if any file uses the .res file name extension, Visual Basic interprets that it is a resource file when adding it to the project. If the file does not follow the standard format for a resource file, Visual Basic generates an error message the first time you attempt to use the resource file support functions (LoadResString, LoadResPicture, and LoadResData), or when you try to make an .exe file. Visual Basic will generate the same error message if you try to add a 16-bit resource file to a project.

Once the resource file is added to the project, the .res file will appear in the Project window. Unlike a form or a module, however, you cannot view the resource file in Visual Basic. The file is still considered a standard resource, as if it were created or used by Microsoft Visual C++ and most other Windows-based development tools. When you choose Make *projectname*.exe from the File menu, Visual Basic compiles all resources in this file into the .exe file as Windows resources.

The .res file, before and after you compile the .exe file, is a standard *Windows resource file*, which means the resources contained in the file can be loaded in any standard Windows-based resource editor.

Locking Resource Files

Visual Basic uses file locking on the .res file to prevent problems with multiple applications trying to use the file at the same time. Visual Basic will lock the .res file whenever:

- Visual Basic is in run or break mode.
- You create an .exe file.

8

For More Information For an example of how a resource file can be used to create an application that works in several locales, see "The Automated Teller Machine Sample Application" later in this chapter. For background information about programming with resource files, see "Working with Resource Files" in "More About Programming."

The Automated Teller Machine Sample Application

This sample application has been designed to illustrate support for resource files in Visual Basic. The application contains three forms, a standard module, and a resource file. When you run the Automated Teller Machine (Atm.vbp) sample application, an opening screen lets you perform a bank transaction in one of several languages, including German, French, Italian, and Spanish.

The following code from the FrmInput.frm file loads resources stored in the Atm32.res file, which contains the localized strings for all languages.

```
Sub Form_Load()
    imgFlag = LoadResPicture(I, vbResBitmap)
    Caption = LoadResString(I)
    lblPINCode = LoadResString(1 + I)
    fraAccount = LoadResString(2 + I)
    optChecking.Caption = LoadResString(3 + I)
    optSavings.Caption = LoadResString(4 + I)
    lblAmount = LoadResString(5 + I)
    cmdOK.Caption = LoadResString(6 + I)
    SetCursor cmdOK
End Sub

Sub cmdOK_click()
    ' Display a process message.
    MsgBox LoadResString(7 + I)
    frmAmountWithdrawn.Show vbModal
    Unload Me
End Sub
```

At run time, this code reads the appropriate section of the resource file, based on an offset that is initialized when the user makes a language selection in the opening screen. The *offset* is a public variable declared in the standard module that indicates how far from a starting point a particular item is located. In the ATM sample application, the offset variable is I.

In the resource file, resource identifiers 16 through 47 are reserved for English, 48 through 79 are reserved for French, 80 through 111 are reserved for German, and so on. Each language contains the localized entries that make up the data block of the sample application. This block currently contains the eleven resources that are particular to each language.

This sample application, which contains several data blocks, introduces an alternative to a language-specific resource file using only one data block. Depending on the nature of the application you are developing, you may consider using one resource file per language version of your application or a single resource file containing all the localized data blocks.

The design of the Automated Teller Machine sample application presents several advantages beyond the ones outlined earlier in the chapter:

- The application can grow in scope by providing service in more languages. Simply add the same data block to the resource file and localize it as needed. If you decide to add a language, you may have to add a button to the opening screen.
- The application can grow in size if you want to extend your application by, for instance, allowing the ATM users to make deposits. Simply allow for wider identifier ranges (160 for example) for each language in the resource file. Currently, the identifiers range from 16 to 47, 48 to 79, and so on.

For More Information For information on resource files, see "Working with Resource Files" in "More About Programming."

Designing an International-Aware User Interface

Because text tends to grow when you localize an application, you should pay special attention when designing the following user interface (UI) components:

- Messages
- Menus and dialog boxes
- Icons and bitmaps
- Access and shortcut keys

10

Messages

English text strings are usually shorter than localized text strings in other languages. The following table shows the additional average growth for strings, based on initial length. This data is drawn from past Visual Basic localization projects and describes an average growth rate.

English length (in characters)	Additional growth for localized strings
1 to 4	100%
5 to 10	80%
11 to 20	60%
21 to 30	40%
31 to 50	20%
over 50	10%

7

When designing the interface, consider these growth ratios and allow for text to wrap to more lines as the messages get longer.

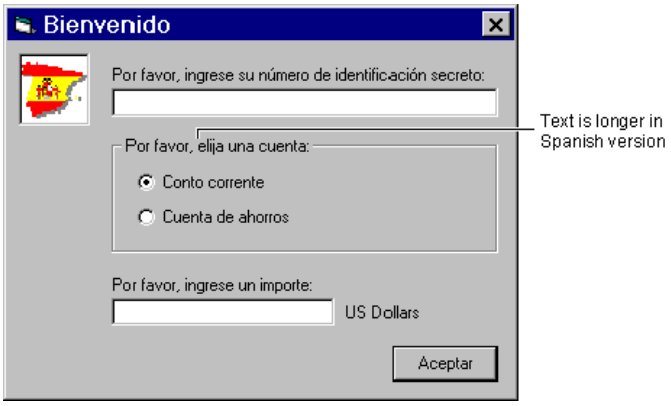
Menus and Dialog Boxes

As with messages, menus and dialog boxes may grow when the application is localized. Consider the two following identical dialog boxes in the Automated Teller Machine sample application. You can see that extra space was allocated in the dialog box to allow for text expansion. Figure 16.2 shows the English dialog box, while Figure 16.3 shows the Spanish dialog box. Knowing that text can grow, plan your interface so that controls don't have to be resized or other elements redesigned when localized.

Figure 16.2 English input dialog box in the ATM sample application

11

Figure 16.3 Spanish input dialog box in the ATM sample application



12

In menus and dialog boxes, avoid crowding status bars. Even abbreviations might be longer or simply not exist in other languages.

Icons and Bitmaps

Icons and bitmaps are usually used to depict a certain functionality without using text. Consider the following rules when working with icons and bitmaps:

- Avoid using bitmaps that are not an international standard. The following bitmaps represent a mailbox in the United States, but many users from other locales will not recognize it.

1  

13

- Avoid using bitmaps that contain text. They take time to redraw, and text growth might also become an obstacle, as illustrated in the following icons.

2 

14

- Make sure that bitmaps or icons are culturally sensitive. What may be acceptable in one locale may be inappropriate or offensive in another.

15

Access and Shortcut Keys

Different locales have different keyboard layouts. Not all characters exist in all keyboard layouts. When developing your Visual Basic application, make sure all access-key and shortcut-key combinations you assign can be reproduced with international keyboards. One simple method to verify that your keyboard assignments work properly for the locales you are targeting is to choose the desired keyboard layout from your Windows Control Panel, along with keyboard layout pictures (which some reference manuals contain), and try the access-key and shortcut-key combinations.

Because certain access-key and shortcut-key combinations are not available for certain locales or because they are reserved for system use by some editions of Windows, it is best to avoid them when developing your Visual Basic application. Here are some examples of characters to avoid:

@ $ { } [ ] \ ~ | ^ ' < >

One way to work around this limitation is to use numbers and function keys (F1, F2, etc.) instead of letters in shortcut-key combinations. These may be less intuitive but they will not require any localization, because virtually all keyboard layouts include numbers and function keys.

Note DBCS characters cannot be used as access or shortcut keys.

8

General Considerations When Writing International Code

When you're developing an application that will be localized — whether you're programming with Visual Basic or another tool — you must take into account differences between languages. You should identify the strings that need to be localized, allow for strings to grow, and avoid the pitfalls of string concatenation.

Hard-Coded Localizable Strings

The localization model presented in "Designing International Software" introduced the concepts of data block and code block. When building the resource files containing all the localizable strings, it is important to include *only* the strings that need to be localized. Any item that does not need to be partially or entirely localized can be hard-coded. Conversely, it is also fundamental to make sure all the resources that need to be localized are actually present in these resource files.

Buffer Sizes

If you are declaring a buffer size based on the expected length of a string or word, make sure this buffer can accommodate larger words and strings. See "Designing an International-Aware User Interface" for average growth rates of translated strings. The buffer size you declare in your Visual Basic code must account for this increase.

Consider the following example. Your Visual Basic declares a 2-byte buffer size for the word "OK." In Spanish, however, the same word is translated as "Aceptar," which would cause your program to overflow. Identical considerations apply for double-byte characters. Refer to "Issues Specific to the Double-Byte Character Set (DBCS)" later in this chapter for more information about DBCS.

String Concatenation

When you try to reduce the size of a string, one possible approach is string concatenation. This method allows you to use the same resource in several strings. However, there are some dangers when using this approach. Consider the following example:

English	French
String1: one after the other	String1: un après l'autre
String2: The controls will be deleted.	String2: Les contrôles seront supprimés.
String3: The forms will be deleted.	String3: Les feuilles seront supprimées.

9

Taken separately, String1, String2, and String3 can be easily localized. If your code performs String2 + String1 or String3 + String1, the resulting English string would look fine. The localized strings, however, are likely to be wrong. In the French column, for instance, String3 + String1 would be wrong because in French grammar, forms (feuilles) is a feminine word, thus String1 should be "une après l'autre" and not "un après l'autre." The same situation will be true in many other foreign languages. The only way to avoid this is to keep String2 and String1, and String3 and String1, together in the resource file.

In the above example, the order of the words that make up the sentence is the same in English and in French. However, the order is generally not the same in these two languages, or many other foreign languages. (For example, in both German and Japanese the verb generally appears at the end of the sentence.) The following example illustrates this situation:

English	French
String1: DLL	String1: DLL
String2: Missing Data Access	String2: Accès aux données manquante
String3: Missing OLE	String3: OLE manquante

10

If your code performs String2 + String1 and String3 + String1, localized versions will be broken because the order of the two strings produces a message that does not make any sense. One possible solution is to simply add String1 to String2 and String3 directly in the resource file and remove String1.

Another possible solution is presented in the following table:

English	French
String2: Missing Data Access '|1'	String2: '|1' d'accès aux données manquant
String3: Missing OLE '|1'	String3: '|1' OLE manquant

11

In this case, the localizer can identify '|1' as a placeholder and make the necessary changes in the resource file to reflect the appropriate way to build a sentence for the localized language.

Finally, it is also important to know that words or sentences that appear identical in English may need to be translated into different words or sentences when localized. Consider the following example:

English	French
String1: Setup program	String1: Programme d'installation
String2: String1 did not complete successfully.	String2: String1 a échoué.

12

In the English version, String1 is used as the setup program banner. It is also used as part of an error message in String2. In the French version, String1 worked perfectly as the stand-alone banner string. However, it needed to become "Le programme d'installation" to be used with String2.

Writing International Code in Visual Basic

Preparing a product for use in other locales implies more than just translating text messages. The product must support national conventions and provide country-specific support for numbers. In order to know how to work with different dates, currencies, and numeric values and separators, you have to understand the distinction Visual Basic makes between system locale and code locale.

System Locale vs. Code Locale

The *system locale* is the locale of the user who runs your program — it is used as a reference for user input and output and uses Control Panel settings provided by the operating system. The *code locale* is always English/U.S. in Visual Basic 5.0,

regardless of which international version you use. Code locale determines the programming language and all the locale-specific settings.

Date

In Visual Basic, never type dates as strings in your code. Entering dates in code in the format #month/day/year# ensures that the date will be interpreted correctly in any system locale. Because Visual Basic allows only English/U.S. as a programming locale, the date will be the same to a user wherever your application is run.

For example, if a user enters 8/2/97 in an input dialog box,

```
CDate ("8/2/97")
```

13

returns the following results, based on the system locale:

Operating system	Output
French/France	08/02/97 (= February 8, 1997)
English/U.S.	8/2/97 (= August 2, 1997)

14

Conversely, if you enter 8/2/97 in code,

```
CDate (#8/2/97#)
```

15

returns the results in the following table, based on the code locale:

Operating system	Output
French/France	02/08/97 (= August 2, 1997)
English/U.S.	8/2/97 (= August 2, 1997)

16

If the user is in France and enters 8/2/97, the application will interpret this date as February 8, 1997, because the date format in France is day/month/year. If a user in the United States enters the same string, the application will understand August 2, 1997, because the date format is month/day/year.

Currency

Avoid typing currencies as strings in your code. For example, the following code does not run in any locale except those where the dollar sign ($) is the currency symbol.

```
Money = "$1.22"
NewMoney = CCur(Money)
```

17

If you run this code example in the French/France locale, where "F" is the currency symbol, Visual Basic will generate a "Type mismatch" error message, because $ is not recognized as a currency symbol in that locale. Instead, simply use numbers, as shown in the following example. Use a period as a decimal separator, because the code locale for Visual Basic is always English/U.S. The following code will run correctly, regardless of the user's locale.

```
Money = 1.22
NewMoney = CCur(Money)
```

18

Numeric Values and Separators

In the United States, the period (.) is used as the decimal separator. In several European countries, however, the comma (,) is used as the decimal separator. Similarly, in the United States, a comma is used as the thousands separator to isolate groups of three digits to the left of the decimal separator. In several European countries, a period or a space is used for this purpose. The following table lists some examples of different number formats:

Countries	Number formats
U.S.	1,234,567.89 1,234.56 .123
France	1 234 567,89 1 234,56 0,123
Italy	1.234.567,89 1.234,56 0,123

19

Note In Visual Basic, the Str and Val functions always assume a period is the decimal separator. In a majority of locales, this assumption is not valid. Instead, use the CStr, CDbl, CSng, CInt, and CLng functions to provide international conversions from any other data type to the data type you need. These functions use the system locale to determine the decimal separator.

20

For More Information See "Locale-Aware Functions" later in this chapter for more information about the Print and Format functions. See also "Variables, Constants, and Data Types."

Locale-Aware Functions

Each locale has different conventions for displaying dates, time, numbers, currency, and other information. It is not necessary to know all the conventions of your users' locales. In Visual Basic, many functions use the user's system locale, which uses the Control Panel settings provided by the operating system to automatically determine the conventions at run time. These functions are called *locale-aware* functions.

Print Function

Even though the Print function provides little flexibility for different output formats, it does use the user's system locale. In the following example, dates are printed using the correct short date format, numbers are printed with the correct decimal separator, and currencies are printed with the correct symbol:

```
MyDate = #11/24/1997#
MyNumber = 26.5
```

```
Money = 1636.32
MyMoney = Format(Money, "###,###.##")
Debug.Print MyDate, MyNumber, MyMoney
```

21

When this code is run in an English/U.S. locale, the following output appears in the Immediate window:

```
11/24/1997 26.5      1,636.32
```

22

When this code is run in a German/Germany locale, the following output appears in the Immediate window:

```
24/11/1997 26,5      1.632,32
```

23

Format Function

The Format function can accept format codes, but format codes always produce the same type of output regardless of the user's locale. For example, the format code "mm-dd-yy" is not appropriate for a user in Belgium, where the day precedes the month.

For more flexibility, the Format function also provides named formats that will automatically determine which conventions to use at run time, including General Date, Long Date, Short Date, and Long Time. Using named formats produces output that is based on the user's system locale. The named formats can even generate output in the user's native language, including the names of months and days of the week. The following example illustrates this:

```
MyDate = #8/22/1997 5:22:20 PM#
NewDate1 = Format(MyDate, "Medium Date")
NewDate2 = Format(MyDate, "Short Date")
NewDate3 = Format(MyDate, "Long Date")
NewDate4 = Format(MyDate, "General Date")
Debug.Print NewDate1, NewDate2, NewDate3, NewDate4
```

24

When this code is run in an English/U.S. locale, the following output appears in the Immediate window:

```
22-Aug-97 8/22/97       Monday, August 22, 1997 8/22/97 5:22:20 PM
```

25

When this code is run in a French/France locale, the following output appears in the Immediate window:

```
22-août-97  22/08/97        lundi 22 août 1997          22/08/97 17:22:20
```

26

International Sort Order and String Comparison

String comparison is widely used in Visual Basic. Using this functionality, however, may yield incorrect results if you overlook certain programming requirements.

Sorting Text

Sorting text means ordering text according to language conventions. Format and font are irrelevant to the sorting process because both involve presentation rather than

content. At first glance, sorting text looks simple: *a* precedes *b*, *b* precedes *c*, and so on. However, there are many languages that have more complex rules for sorting. Correct international sorting is not always a simple extension of sorting English text, and it requires a different understanding of the sorting process.

Correct international sorting can imply *context-sensitive sorting*. Character contraction and expansion are the two important areas of context-sensitive sorting.

- *Character contraction* occurs when a two-character combination is treated as a single, unique letter. For example, in Spanish the two-character combination *ch* is a single, unique letter and sorts between *c* and *d*.
- *Character expansion* occurs in cases where one letter represents one character, but that one character sorts as if it were two. For example, *ß* (eszett) is equivalent to *ss* in both German/Germany and German/Switzerland locales. However, *ß* is equivalent to *sz* in the German/Austria locale.

16

Before implementing the sorting order, you must consider code pages. A *code page* is an ordered character set that has a numeric index (code point) associated with each character. Because there are various code pages, a single code point might represent different characters in different code pages. While most code pages share the code points 32 through 127 (ASCII character set), they differ beyond that. Typically, the ordering of any additional letters in these code pages is not alphabetic.

For More Information See "DBCS Sort Order and String Comparison" later in this chapter for more information about working with East Asian languages.

27

String Comparison in Visual Basic

String comparison rules are different for each locale. Visual Basic provides a number of tools, such as Like and StrComp, which are locale-aware. To use these effectively, however, the Option Compare statement must first be clearly understood.

Comparing Strings with the Option Compare Statement

When using this statement, you must specify a string comparison method: either Binary or Text for a given module. If you specify Binary, comparisons are done according to a sort order derived from the internal binary representations of the characters. If you specify Text, comparisons are done according to the case-insensitive textual sort order determined by the user's system locale. The default text comparison method is Binary.

In the following code example, the user enters information into two input boxes. The information is then compared and sorted in the appropriate alphabetic order.

```
Private Sub Form_Click ()
Dim name1 As String, name2 As String
    name1 = InputBox("Enter 1st hardware name here:")
    name2 = InputBox("Enter 2nd hardware name here:")
If name1 < name2 Then
    msg = " ' " & name1 & " ' comes before ' " & _
```

—16

```
    name2 & " ' "
Else
    msg = " ' " & name2 & " ' comes before ' " & _
    name1 & " ' "
End If
    MsgBox msg
End Sub
```

28

If this code is run in an English/U.S. locale, the message box will contain the following output if the user enters printer and Screen:

```
'Screen' comes before 'printer'
```

29

This result is based on the fact that the default text-comparison method is Binary. Because the internal binary representation of uppercase *S* is smaller than the one for lowercase *p*, the conditional statement Screen < printer is verified. When you add the Option Compare Text statement in the Declarations section of a module, Visual Basic compares the two strings on a case-insensitive basis, resulting in the following output:

```
'printer' comes before 'Screen'
```

30

If this code is run in a French/Canada locale, the message box will contain the following output if the user enters imprimante and écran:

```
'imprimante' comes before 'écran'
```

31

Similarly, if you add the Option Compare Text statement to your code, the two terms will appear in the right order — that is, écran will precede imprimante. In addition to being case insensitive, the comparison takes into account the accented characters, such as *é* in French, and places it right after its standard character — in this case, *e*, in the sorting order.

If the user had entered ecran and écran, the output would be:

```
'ecran' comes before 'écran'
```

32

Comparing Strings with the Like Operator

You can use the Like operator to compare two strings. You can also use its pattern-matching capabilities. When you write international software, you must be aware of pattern-matching functions. When character ranges are used with Like, the specified pattern indicates a range of the sort ordering. For example, under the Binary method for string comparison (by default or by adding Option Compare Binary to your code), the range [A – C] would miss both uppercase accented *a* characters and all lower-case characters. Only strings starting with A, B, and C would match. This would not be acceptable in many languages. In German, for instance, the range would miss all the strings beginning with *Ä*. In French, none of the strings starting with *À* would be included.

Under the Text method for string comparison, all the accented *A* and *a* characters would be included in the interval. In the French/France locale, however, strings starting with *Ç* or *ç* would not be included, since *Ç* and *ç* appear after *C* and *c* in the sort order.

Using the [A – Z] range to check for all strings beginning with an alphabetic character is not a valid approach in certain locales. Under the Text method for string comparison, strings beginning with *Ø* and *ø* would not be included in the range if your application is running in a Danish/Denmark locale. Those two characters are part of the Danish alphabet, but they appear after *Z*. Therefore, you would need to add the letters after Z. For example, Print "øl" Like "[A-Z]*" would return False, but Print "øl" Like "[A-ZØ]*" would return True with the Option Compare Text statement.

Comparing Strings with the StrComp Function

The StrComp function is useful when you want to compare strings. It returns a value that tells you whether one string is less than, equal to, or greater than another string. The return value is also based on the string comparison method (Binary or Text) you defined with the Option Compare statement. StrComp may give different results on the strings you compare, depending on the string comparison method you define.

For More Information See "DBCS Sort Order and String Comparison" later in this chapter for more information about comparing strings in East Asian languages.

International File Input/Output

Locale is also an important consideration when working with file input and output in Visual Basic. Both the Print # and Write # statements can be used to work with data files, but they have distinct purposes.

Print

The Print # statement puts data into a file as the data is displayed on the screen, in a locale-aware format. For instance, date output uses the system Short Date format, and numeric values use the system decimal separator.

The Input # statement cannot read locale-aware data in Visual Basic that has been written to a file with the Print # statement. To write locale-independent data that can be read by Visual Basic in any locale, use the Write # statement instead of the Print # statement.

Write

Like the Print # statement, the Write # statement puts data into a file in a fixed format, which ensures that the data can be read from the file in any locale when using the Input # statement. For instance, dates are written to the file using the universal date format, and numeric values are written to the file using the period as the decimal separator. In the following code example, a date and a numeric value are written to a file with the Write # statement. The same file is reopened later, its content is read with the Input # statement, and the results are printed in the Immediate window. The Long Date information is drawn from the system locale:

```
Dim MyDate As Date, NewDate As Date
Dim MyNumber As Variant
    MyDate = #8/2/67#
```

```
    MyNumber = 123.45
Open "Testfile" for Output As #1
    Write #1, MyDate, MyNumber
Close #1

Open "Testfile" for Input As #1
    Input #1, MyDate, MyNumber
    NewDate = Format(Mydate, "Long Date")
Debug.Print NewDate, MyNumber
Close #1
```

33

When you run this code in an English/U.S. locale, the following output appears in the Immediate window:

```
Wednesday, August 02, 1967          123.45
```

34

When you run this code in a French/France locale, the following output appears in the Immediate window:

```
mercredi 2 août 1967                123,45
```

35

In both locales, the output is accurate — that is, the information was stored and retrieved properly using the Write # and Input # statements.

17

Locale-Aware SQL Queries Based on Dates

As explained in "Writing International Code in Visual Basic," different countries have different date formats. If your application performs a comparison between two dates, date literals must be stored in a unique format to ensure a reliable comparison, regardless of a user's locale. In Visual Basic, the database engine stores a date/time value as a DateSerial value, which is represented by an 8-byte floating-point number, with the date as the integral portion and the time as the fractional portion. This approach is completely locale-independent and will let you perform date/time comparisons using the international date/time formats.

Structured Query Language (SQL) is an ANSI standard with which Visual Basic complies. Dates are saved in tables and databases using the English/U.S. format (month/day/year). This format was also adopted for the Microsoft Jet database engine. Queries that use these fields may return the wrong records or no records at all if a non-U.S. date format is used.

This constraint also applies to the Filter property, to the FindFirst, FindNext, FindPrevious, and FindLast methods of the Recordset object, and to the WHERE clause of an SQL statement.

Using DateSerial and DateValue

There are two functions you can use to handle the limitations of the SQL standard. Avoid using date/time literals in your code. Instead, consider using the DateValue or the DateSerial functions to generate the date you want. The DateValue function uses the system's Short Date setting to interpret the string you supply; the DateSerial

function uses a set of arguments that will run in any locale. If you are using date/time literals in your SQL query or with the Filter property, you have no choice but to use the English/U.S. format for date and time.

The following examples illustrate how to perform a query based on a date. In the first example, a non-U.S. date format is used. The Recordset returned is empty because there is a syntax error in the date expression:

```
Dim mydb As Database
Dim myds As Recordset

Set mydb = OpenDatabase("MyDatabase.mdb")
' Table that contains the date/time field.
Set myds = mydb.OpenRecordset("MyTable,dbopenDynaset")
' The date format is dd/mm/yy.
myds.FindFirst "DateFiled > #30/03/97#"
' A data control is connected to mydb.
Data1.Recordset.Filter = "DateFiled = #30/03/97#"

mydb.Close
myds.Close
```

36

The following example, however, will work adequately in any locale because the date is in the appropriate format:

```
Dim mydb As Database
Dim myds As Recordset

Set mydb = OpenDatabase("MyDatabase.mdb")
' Table that contains the date/time field.
Set myds = mydb.OpenRecordset("MyTable, dbopenDynaset")

myds.FindFirst "DateFiled > #03/30/97#"' Date format
                                                    ' is mm/dd/yy.

' A data control is connected to mydb.
Data1.Recordset.Filter = "DateFiled = _
DateValue(""" & DateString & """)"

mydb.Close
myds.Close
```

37

Issues Specific to the Double-Byte Character Set (DBCS)

The *double-byte character set* (DBCS) was created to handle East Asian languages that use ideographic characters, which require more than the 256 characters supported by ANSI. Characters in DBCS are addressed using a 16-bit notation, using 2 bytes. With 16-bit notation you can represent 65,536 characters, although far fewer

characters are defined for the East Asian languages. For instance, Japanese character sets today define about 12,000 characters.

In locales where DBCS is used — including China, Japan, Taiwan, and Korea — both single-byte and double-byte characters are included in the character set. The single-byte characters used in these locales conform to the 8-bit national standards for each country and correspond closely to the ASCII character set. Certain ranges of codes in these single-byte character sets (SBCS) are designated as *lead bytes* for DBCS characters. A consecutive pair made of a lead byte and a trail byte represents one double-byte character. The code range used for the lead byte depends on the locale.

Note DBCS is a different character set from Unicode. Because Visual Basic represents all strings internally in Unicode format, both ANSI characters and DBCS characters are converted to Unicode and Unicode characters are converted to ANSI characters or DBCS characters automatically whenever the conversion is needed. You can also convert between Unicode and ANSI/DBCS characters manually. For more information about conversion between different character sets, see "DBCS String Manipulation Functions."

38

When developing a DBCS-enabled application with Visual Basic, you should consider:

- Differences between Unicode, ANSI, and DBCS.
- DBCS sort orders and string comparison.
- DBCS string manipulation functions.
- DBCS string conversion.
- How to display and print fonts correctly in a DBCS environment.
- How to process files that include double-byte characters.
- DBCS identifiers.
- DBCS-enabled events.
- How to call Windows APIs.

18

Tip Developing a DBCS-enabled application is good practice, whether or not the application is run in a locale where DBCS is used. This approach will help you develop a flexible, portable, and truly international application. None of the DBCS-enabling features in Visual Basic will interfere with the behavior of your application in environments using exclusively single-byte character sets (SBCS), and the size of your application will not increase because both DBCS and SBCS use Unicode internally.

39

For More Information For limitations on using DBCS for access and shortcut keys, see "Designing an International-Aware User Interface."

40

ANSI, DBCS, and Unicode: Definitions

Visual Basic uses Unicode to store and manipulate strings. Unicode is a character set where 2 bytes are used to represent each character. Some other programs, such as the Windows 95 API, use ANSI (American National Standards Institute) or DBCS to store and manipulate strings. When you move strings outside of Visual Basic, you may encounter differences between Unicode and ANSI/DBCS. The following table shows the ANSI, DBCS, and Unicode character sets in different environments.

Environment	Character set(s) used
Visual Basic	Unicode
32-bit object libraries	Unicode
16-bit object libraries	ANSI and DBCS
Windows NT API	Unicode
Automation in Windows NT	Unicode
Windows 95 API	ANSI and DBCS
Automation in Windows 95	Unicode

41

ANSI

ANSI is the most popular character standard used by personal computers. Because the ANSI standard uses only a single byte to represent each character, it is limited to a maximum of 256 character and punctuation codes. Although this is adequate for English, it doesn't fully support many other languages.

DBCS

DBCS is used in Microsoft Windows systems that are distributed in most parts of Asia. It provides support for many different East Asian language alphabets, such as Chinese, Japanese, and Korean. DBCS uses the numbers 0 – 128 to represent the ASCII character set. Some numbers greater than 128 function as *lead-byte characters*, which are not really characters but simply indicators that the next value is a character from a non-Latin character set. In DBCS, ASCII characters are only 1 byte in length, whereas Japanese, Korean, and other East Asian characters are 2 bytes in length.

Unicode

Unicode is a character-encoding scheme that uses 2 bytes for *every* character. The International Standards Organization (ISO) defines a number in the range of 0 to 65,535 ($2^{16} – 1$) for just about every character and symbol in every language (plus some empty spaces for future growth). On all 32-bit versions of Windows, Unicode is used by the Component Object Model (COM), the basis for OLE and ActiveX technologies. Unicode is fully supported by Windows NT. Although both Unicode and DBCS have double-byte characters, the encoding schemes are completely different.

Character Code Examples

Figure 16.4 shows an example of the character code in each character set. Note the different codes in each byte of the double-byte characters.

Figure 16.4 Character codes for "A" in ANSI, Unicode, and DBCS

A	ANSI character "A"	&H41
A	Unicode character "A"	&H41 &H00
A	DBCS character that represents a Japanese wide-width "A"	&H82 &H60
A	Unicode wide-width "A"	&H21 &HFF

DBCS Sort Order and String Comparison

You need to be aware of the issues when sorting and comparing DBCS text, because the Option Compare Text statement has a special behavior when used on DBCS strings. When you use the Option Compare Binary statement, comparisons are made according to a sort order derived from the internal binary representations of the characters. When you use Option Compare Text statement, comparisons are made according to the case-insensitive textual sort order determined by the user's system locale.

In English "case-insensitive" means ignoring the differences between uppercase and lowercase. In a DBCS environment, this has additional implications. For example, some DBCS character sets (including Japanese, Traditional Chinese, and Korean) have two representations for the same character: a narrow-width letter and a wide-width letter. For example, there is a single-byte "A" and a double-byte "A." Although they are displayed with different character widths, Option Compare Text treats them as the same character. There are similar rules for each DBCS character set.

You need to be careful when you compare two strings. Even if the two strings are evaluated as the same using Like or StrComp, the exact characters in the strings can be different and the string length can be different, too.

For More Information For general information about comparing strings with the Option Compare statement, see "International Sort Order and String Comparison."

DBCS String Manipulation Functions

Although a double-byte character consists of a lead byte and a trail byte and requires two consecutive storage bytes, it must be treated as a single unit in any operation involving characters and strings. Several string manipulation functions properly handle all strings, including DBCS characters, on a character basis.

These functions have an ANSI/DBCS version and a binary version and/or Unicode version, as shown in the following table. Use the appropriate functions, depending on the purpose of string manipulation.

The "B" versions of the functions in the following table are intended especially for use with strings of binary data. The "W" versions are intended for use with Unicode strings.

Function	Description
Asc	Returns the ANSI or DBCS character code for the first character of a string.
AscB	Returns the value of the first byte in the given string containing binary data.
AscW	Returns the Unicode character code for the first character of a string.
Chr	Returns a string containing a specific ANSI or DBCS character code.
ChrB	Returns a binary string containing a specific byte.
ChrW	Returns a string containing a specific Unicode character code.
Input	Returns a specified number of ANSI or DBCS characters from a file.
InputB	Returns a specified number of bytes from a file.
InStr	Returns the first occurrence of one string within another.
InStrB	Returns the first occurrence of a byte in a binary string.
Left, Right	Returns a specified number of characters from the right or left sides of a string.
LeftB, RightB	Returns a specified number of bytes from the left or right side of a binary string.
Len	Returns the length of the string in number of characters.
LenB	Returns the length of the string in number of bytes.
Mid	Returns a specified number of characters from a string.
MidB	Returns the specified number of bytes from a binary string.

42

The functions without a "B" or "W" in this table correctly handle DBCS and ANSI characters. In addition to the functions above, the String function handles DBCS characters. This means that all these functions consider a DBCS character as one character even if that character consists of 2 bytes.

The behavior of these functions is different when they're handling SBCS and DBCS characters. For instance, the Mid function is used in Visual Basic to return a specified number of characters from a string. In locales using DBCS, the number of *characters* and the number of *bytes* are not necessarily the same. Mid would only return the number of characters, not bytes.

In most cases, use the character-based functions when you handle string data because these functions can properly handle ANSI strings, DBCS strings, and Unicode strings.

The byte-based string manipulation functions, such as LenB and LeftB, are provided to handle the string data as binary data. When you store the characters to a String variable or get the characters from a String variable, Visual Basic automatically converts between Unicode and ANSI characters. When you handle the binary data, use the Byte array instead of the String variable and the byte-based string manipulation functions.

If you want to handle strings of binary data, you can map the characters in a string to a Byte array by using the following code:

```
Dim MyByteString() As Byte
' Map the string to a Byte array.
MyByteString = "ABC"
' Display the binary data.
For i = LBound(MyByteString) to UBound(MyByteString)
    Print Right(" " + Hex(MyByteString(i)),2) + " ,";
Next
Print
```


DBCS String Conversion

Visual Basic provides several string conversion functions that are useful for DBCS characters: StrConv, UCase, and LCase.

StrConv Function

The global options of the StrConv function are converting uppercase to lowercase, and vice versa. In addition to those options, the function has several DBCS-specific options. For example, you can convert narrow letters to wide letters by specifying vbWide in the second argument of this function. You can convert one character type to another, such as hiragana to katakana in Japanese.

You can also use the StrConv function to convert Unicode characters to ANSI/DBCS characters, and vice versa. Usually, a string in Visual Basic consists of Unicode characters. When you need to handle strings in ANSI/DBCS (for example, to calculate the number of bytes in a string before writing the string into a file), you can use this functionality of the StrConv function.

Case Conversion in Wide-Width Letters

You can convert the case of letters by using the StrConv function with vbUpperCase or vbLowerCase, or by using the UCase or LCase functions. When you use these functions, the case of English wide-width letters in DBCS are converted as well as ANSI characters.

Font, Display, and Print Considerations in a DBCS Environment

When you use a font designed only for SBCS characters, DBCS characters may not be displayed correctly in the DBCS version of Windows. You need to change the Font object's Name property when developing a DBCS-enabled application with the English version of Visual Basic or any other SBCS-language version. The Name property determines the font used to display text in a control, in a run-time drawing, or during a print operation. The default setting for this property is MS Sans Serif in the English version of Visual Basic. To display text correctly in a DBCS environment, you have to change the setting to an appropriate font for the DBCS environment where your application will run. You may also need to change the font size by changing the Size property of the Font object. Usually, the text in your application will be displayed best in a 9-point font on most East Asian platforms, whereas an 8-point font is typical on European platforms.

These considerations apply to printing DBCS characters with your application as well.

How to Avoid Changing Font Settings

If you do not have any DBCS-enabled font or do not know which font is appropriate for the target platform, there are several options for you to work around the font issues.

In the Traditional Chinese, Simplified Chinese, and Korean versions of Windows, there is a system capability called *Font Association*. With Korean Windows, for example, Font Association automatically maps any English fonts in your application to a Korean font. Therefore, you can still see Korean characters displayed, even if your application uses English fonts. The associated font is determined by the setting in \HKEY_LOCAL_MACHINE\System\CurrentControlSet\control\fontassoc \Associated DefaultFonts in the system registry of the run-time platform. With Font Association supported by the system, you can run your English application on a Chinese or Korean platform without changing any font settings. Font Association is not available on other platforms, such as Japanese Windows.

Another option is to use the System or FixedSys font. These fonts are available on every platform. Note that the System and FixedSys fonts have few variations in size. If the font size you set at design time (with the Size property of the Font object) for either of these fonts does not match the size of the font on the user's machine, the setting may be ignored and the displayed text truncated.

How to Change the Font at Run Time

Even though you have the options above, these solutions have restrictions. Here is an example of a global solution to changing the font in your application at run time. The following code, which will work on any language version of Windows, determines a font that resides in the system where the application is running and applies that font to your application's form.

```
Private Declare Function GetStockObject Lib "gdi32" _
(ByVal nIndex As Long) As Long
```

```
Private Declare Function SelectObject Lib "gdi32" _
(ByVal hdc As Long, ByVal hObject As Long) As Long
Private Declare Function GetTextFace Lib "gdi32" _
Alias "GetTextFaceA" (ByVal hdc As Long, _
ByVal nCount As Long, ByVal lpFacename As _
String) As Long
Private Declare Function ReleaseDC Lib "user32" _
(ByVal hwnd As Long, ByVal hdc As Long) As Long

Dim FontFaceName As String

Const DEFAULT_GUI_FONT = 17

Private Sub Form_Load()
' This procedure gets the stock font in the system.
' Stock font is the font used for the user interface
' of Windows. This code should be put into the Form
' module because it requires hWnd and hDc.

    Dim GuiFont As Long, OldFont As Long, Ret As Long
    Dim ctl As Control

    ' Buffer for FontName.
    FontFaceName = Space(80)

    ' Get font handle for DEFAULT_GUI_FONT.
    GuiFont = GetStockObject(DEFAULT_GUI_FONT)
    ' Set GuiFont to the current Window.
    OldFont = SelectObject(Me.hdc, GuiFont)
    ' Get fontface name which will be returned
    ' into FontFaceName.
    Ret = GetTextFace(Me.hdc, 80, FontFaceName)
    ' The following line is required because
    ' FontFaceName is converted to Unicode while
    ' Ret returns ANSI/DBCS length.
    FontFaceName = Left(FontFaceName, InStr_
    (FontFaceName, Chr(0)) - 1)

    Ret = SelectObject(Me.hdc, OldFont)
    Ret = ReleaseDC(Me.hwnd, Me.hdc)            ' Release the
                                                ' object.

    ' Apply this fontface so that the characters on
    ' the form will be displayed correctly.
    Me.FontName = FontFaceName
    On Error Resume Next
    For Each ctl In Controls
        ' If the control does not have Font property,
        ' this line will be skipped.
        ctl.FontName = FontFaceName
    Next
    On Error GoTo 0

End Sub
```

44

You can modify this sample code to apply the font to other font settings, such as printing options.

Processing Files That Use Double-Byte Characters

In locales where DBCS is used, a file may include both double-byte and single-byte characters. Because a DBCS character is represented by two bytes, your Visual Basic code must avoid splitting it. In the following example, assume Testfile is a text file containing DBCS characters.

```
' Open file for input.
Open "TESTFILE" For Input As #1

' Read all characters in the file.
Do While Not EOF(1)
    MyChar = Input(1, #1)    ' Read a character.
    ' Perform an operation using Mychar.
Loop
Close #1                                ' Close file.
```

45

When you read a fixed length of bytes from a binary file, use a Byte array instead of a String variable to prevent the ANSI-to-Unicode conversion in Visual Basic.

```
Dim MyByteString(0 to 4) As Byte

Get #1,, MyByteString
```

46

When you use a String variable with Input or InputB to read bytes from a binary file, Unicode conversion occurs and the result is incorrect.

Keep in mind that the names of files and directories may also include DBCS characters.

For More Information For information on the Byte data type, see "Data Types" in "Programming Fundamentals."

Identifiers in a DBCS Environment

You can use DBCS characters for the following identifiers:

- variable names
- constant names
- procedure names
- object names
- module names, except for class modules
- control names

19

You cannot use DBCS characters for the following identifiers (note that they are not file names, but Visual Basic object identifiers):

- project names (also known as application names)
- class module names

20

Because some identifiers may include DBCS characters, code that uses those names needs to be able to handle DBCS characters correctly. For more information on manipulating DBCS strings, see "DBCS Sort Order and String Comparison" and "DBCS String Manipulation Functions" earlier in this chapter.

DBCS-Enabled KeyPress Event

The KeyPress event can process a double-byte character code as one event. The higher byte of the *keyascii* argument represents the lead byte of a double-byte character, and the lower byte represents the trail byte.

In the following example, you can pass a KeyPress event to a text box, whether the character you input is single-byte or double-byte.

```
Sub Text1_KeyPress (KeyAscii As Integer)
    Mychar = Chr(KeyAscii)
    ' Perform an operation using Mychar.
End Sub
```

47

Calling Windows API Functions

Many Windows API and DLL functions return size in bytes. This return value represents the size of the returned string. Visual Basic converts the returned string into Unicode even though the return value still represents the size of the ANSI or DBCS string. Therefore, you may not be able to use this returned size as the string's size. The following code gets the returned string correctly:

```
buffer = String(145, Chr(" "))
    ret = GetPrivateProfileString(section, _
    entry, default, buffer, Len(buffer)-1, filename)
    retstring = Left(buffer, Instr(buffer, Chr(0))-1)
```

48

For More Information For more information, see "Accessing the Microsoft Windows API" in "Accessing DLLs and the Windows API."