

A P P E N D I X

A

Built-In Functions

A

This appendix describes all Visual Basic built-in functions, grouped by category. Table A.1 lists the individual functions by type. In the following sections, you will find detailed descriptions of each function and some simple examples.

TABLE A.1: Functions by Type

TYPE	FUNCTION
Input/Output	InputBox()
	MsgBox()
File Functions	FileAttr()
	GetAttr()
	FileDateTime()
	FileLen()
	FreeFile()
Variable Type	Array()
	LBound()
	Ubound()
	IsArray()
	IsDate()
	IsEmpty()
	IsNull()
	IsNumeric()
	IsObject()
	TypeName() *
Variable Type Conversion	VarType()
	CBool()
	CByte()
	CCur()
	CDate()
	CDec()

Continued on next page

TABLE A.1 CONTINUED: Functions by Type

TYPE	FUNCTION
Variable Type Conversion	CDbl()
	CInt()
	CLng()
	CSng()
	CStr()
	CVar()
	CVErr()
String-Handling	Asc()
	Chr()
	InStr()
	InStrB()
	Lcase()
	Left()
	Len()
	LTrim()
	Mid()
	Right()
	Space()
	String()
	StrComp()
	StrConv()
	StrReverse()
	Format\$()
	Ucase()
	InStr()
	Str()

Continued on next page

TABLE A.1 CONTINUED: Functions by Type

TYPE	FUNCTION
Data Formatting	Format()
	FormatCurrency()
	FormatDateTime()
	FormatNumber()
	FormatPercent()
Math	Abs()
	Atn()
	Cos()
	Exp()
	Int()
	Fix()
	Round()
	Log()
	Oct()
	Hex()
	Rnd()
	Sgn()
	Sin()
	Sqr()
	Tan()
	Val()
Date and Time	Timer()
	Date()
	Time()
	Now()
	Day()

Continued on next page

TABLE A.1 CONTINUED: Functions by Type

TYPE	FUNCTION
Date andTime	Weekday()
	Month()
	MonthName()
	Year()
	Hour()
	Minute()
	Second()
	DateSerial()
	DateValue()
	TimeSerial()
	TimeValue()
	DateAdd() *
	DateDiff() *
	DatePart() *
Financial	IPmt()
	PPmt()
	Pmt()
	FV()
	PV()
	NPV()
	NPer()
	Rate()
	IRR()
	MIRR()
	DDB()

Continued on next page

TABLE A.1 CONTINUED: Functions by Type

TYPE	FUNCTION
Financial	SYD()
	SLN()
Color	QBColor()
	RGB()
Registry	SaveSettings()
	GetSettings()
	GetAllSettings()
Miscellaneous	Choose()
	Environ()
	IIF()
	Switch()
	Shell()

These functions are described in the following sections, along with examples. The functions are not listed alphabetically within each category. I start with the simpler ones so that I can present examples that combine more than one function.

Input/Output Functions

Visual Basic provides two basic functions for displaying (or requesting) information to the user: `MsgBox()` and `InputBox()`. Windows applications should communicate with the user via nicely designed Forms, but the `MsgBox()` and `InputBox()` functions are still around and quite useful.

InputBox(prompt[, title][, default][, xpos][, ypos][, helpfile, context])

The `InputBox()` function displays a dialog box with a prompt and a `TextBox` control and waits for the user to enter some text and click on the OK or Cancel button. The arguments of the `InputBox()` function are shown in Table A.2.

TABLE A.2: The Arguments of the InputBox() Function

ARGUMENT	WHAT IT IS	DESCRIPTION
prompt	The Prompt that appears in the dialog box	If necessary, the prompt is broken into multiple lines automatically. To control line breaks from within your code, use a carriage return character or a linefeed character (Chr(10)).
title	The title of the dialog box	If you omit this argument, the application's name is displayed as the title.
default	The default input (if any)	If you anticipate the user's response, use this argument to display it when the dialog box is first opened.
xpos, ypos	The coordinates of the upper left corner of the dialog box	Expressed in twips.
helpfile	The name of the Help file	Provides context-sensitive help for the dialog box.
context	The number within the Help file	Assigned to the specific topic.

The simplest format of the InputBox() function is as follows:

```
SSN = InputBox("Please enter your social security number")
```

The string that the user enters in the dialog box is assigned to the variable *SSN*. The return value is always a string, even if the user enters numeric information. When prompting for input with the InputBox() function, always check the value returned by the function. At the very least, check for a blank string. Use the IsNumeric() function if you expect the user to enter a number, use the IsDate() function if you expect the user to enter a date, and so on.

```
BDay = InputBox("Please enter your birth date")
If IsDate(BDay) Then
    MsgBox "Preparing your Horoscope"
Else
    MsgBox "Please try again with a valid birth date"
End If
```

MsgBox(prompt[, buttons][, title] [, helpfile, context]) The MsgBox function displays a dialog box with a message and waits for the user to close it by clicking on a button. The message is the first argument (*prompt*). The simplest form of the MsgBox() function is as follows:

```
MsgBox "Your computer is running out of memory!"
```

This function displays a message in a dialog box that has an OK button. The `MsgBox()` function can display other buttons and/or an icon in the dialog box and return a numeric value, depending on which button was clicked. Table A.3 summarizes the values for the buttons argument.

TABLE A.3: The Values for the Buttons Argument

CONSTANT	VALUE	DESCRIPTION
Button Values		
<code>vbOKOnly</code>	0	Displays OK button only.
<code>VbOKCancel</code>	1	Displays OK and Cancel buttons.
<code>VbAbortRetryIgnore</code>	2	Displays Abort, Retry, and Ignore buttons.
<code>VbYesNoCancel</code>	3	Displays Yes, No, and Cancel buttons.
<code>VbYesNo</code>	4	Displays Yes and No buttons.
<code>VbRetryCancel</code>	5	Displays Retry and Cancel buttons.
Icon Values		
<code>VbCritical</code>	16	Displays Critical Message icon.
<code>VbQuestion</code>	32	Displays Warning Query icon.
<code>VbExclamation</code>	48	Displays Warning Message icon.
<code>VbInformation</code>	64	Displays Information Message icon.
Default Buttons		
<code>VbDefaultButton1</code>	0	First button is default.
<code>VbDefaultButton2</code>	256	Second button is default.
<code>VbDefaultButton3</code>	512	Third button is default.
<code>VbDefaultButton4</code>	768	Fourth button is default.
Modality		
<code>VbApplicationModal</code>	0	Application modal; the user must respond to the message box before switching to any of the Forms of the current application.
<code>VbSystemModal</code>	4096	System modal; all applications are suspended until the user responds to the message box.

Button values determine which buttons appear in the dialog box. Notice that you can't choose which individual buttons to display; you can only choose groups of buttons.

Icon values determine an optional icon you can display in the dialog box. These are the common icons used throughout the Windows user interface to notify the user about an unusual or exceptional event.

Default Button values determine which button is the default one; pressing Enter activates this button

The values 0 and 4096 determine whether the message box is modal. To combine these settings into a single value, simply add their values.

Finally, the `MsgBox()` function returns an integer, which indicates the button pressed, according to Table A.4.

TABLE A.4: The Values of the Buttons

CONSTANT	VALUE	DESCRIPTION
<code>vbOK</code>	1	OK
<code>vbCancel</code>	2	Cancel
<code>vbAbort</code>	3	Abort
<code>vbRetry</code>	4	Retry
<code>vbIgnore</code>	5	Ignore
<code>vbYes</code>	6	Yes
<code>vbNo</code>	7	No

To display a dialog box with the OK and Cancel buttons and the Warning Message icon, add the values 1 and 48 as follows:

```
cont = MsgBox("This operation may take several minutes", 48+1);
```

Your script continues with the operation if the value of `cont` is 1 (OK button), or exit.

To display a dialog box with the Yes and No buttons and the Critical Message icon, add the values 4 and 16 as follows:

```
cont = MsgBox("Incomplete data. Would you like to retry?", 4 + 16);
If cont = 6 Then      // user clicked Yes
    {prompt again}
Else                  // user clicked No
    {exit procedure}
Endif
```

The *title* argument is the title displayed in the message box's title bar. See the description of the `InputBox()` function for an explanation of the *helpfile* and *context* arguments.

File I/O Functions

The following Visual Basic functions manipulate files (create a new file, open an existing file, read and write to a file, examine their properties, and so on).

FileAttr(filename, returntype) The `FileAttr()` function returns a long integer representing the file mode for files opened using the `Open` statement. The *filename* variable is the number of the file, and *returntype* must be 1. The value returned is one of those in Table A.5.

TABLE A.5: The Values Returned by the `FileAttr()` Function

VALUE	MODE
1	Input
2	Output
4	Random
8	Append
32	Binary

GetAttr(filename) This function returns an integer representing the attributes of a file, a directory, or a folder, according to Table A.6.

TABLE A.6: The Values Returned by the `GetAttr()` Function

CONSTANT	VALUE	DESCRIPTION
<code>vbNormal</code>	0	Normal
<code>vbReadOnly</code>	1	Read-only
<code>vbHidden</code>	2	Hidden
<code>vbSystem</code>	4	System
<code>vbDirectory</code>	16	Directory or folder
<code>vbArchive</code>	32	File has changed since last backup

To determine which attributes are set, use the AND operator to perform a bitwise comparison of the value returned by the `GetAttr()` function and the value of one or more attributes. If the result is not zero, that attribute is set for the named file. For example, to find out if a file is read-only, use a statement such as the following:

```
Result GetAttr(FName) And vbReadOnly
```

If the file *Fname* has its read-only attribute set, `Result` will be a read-only file.

FileDateTime(filename) This function returns the date and time when a file was created or last modified. The following statement:

```
Print FileDateTime("myDocument.txt")
```

returns a date/time value such as "21/1/97 14:13:02 PM".

FileLen(filename) The `FileLen()` function returns a long integer value indicating the file's length. The file whose length you want to find out is passed as an argument to the function. The following statement:

```
MsgBox FileLen(".\docs\myDocument.txt")
```

displays the length of the specified file in a message box.

FreeFile() Each file opened with Visual Basic must have a unique handle (a number) that is assigned to the file the moment it's opened and is used to refer to the file in future operations. The handle is freed (that is, it becomes available to be used with another file) after its file is closed.

`FreeFile()` returns an integer representing the next file number available for use by the `Open` statement. Hard-coding a file number is not considered solid programming practice, so you usually call the `FreeFile()` function to find out the next available file number and then open the file using the value returned by the `FreeFile()` function as follows:

```
fileNum = FreeFile  
Open "myDocument.txt" For Output As #fileNum
```

The actual value of the file number is unimportant as long you use the variable *fileNum* to refer to the file `myDocument.txt`.

Variable Type Functions

The following functions manipulate variables. Some functions let you determine a variable's exact type from within your code, and a series of functions determine the general type of a variable (such as numeric, date, and so on). In addition, three functions let you populate array elements and quickly check array bounds.

VarType(variable) The VarType() function returns a value indicating the sub-type of a variable, according to Table A.7.

TABLE A.7: The Values Returned by the VarType() Function

CONSTANT	VALUE	DESCRIPTION
vbEmpty	0	Empty (uninitialized)
vbNull	1	Null (no valid data)
vbInteger	2	Integer
vbLong	3	Long integer
vbSingle	4	Single-precision floating-point number
vbDouble	5	Double-precision floating-point number
vbCurrency	6	Currency value
vbDate	7	Date value
vbString	8	String
vbObject	9	Object
vbError	10	Error value
vbBoolean	11	Boolean value
vbVariant	12	Variant (used only with arrays of variants)
vbDataObject	13	A data access object
vbDecimal	14	Decimal value
vbByte	17	Byte value
vbArray	8192	Array

The VarType() function doesn't return the type of an array's elements directly. Instead, the value of the elements' type is added to 8,192. If you pass an array of strings to the VarType() function, the return value is 8,200 (which is 8,192 + 8).

Array(argumentList) This function returns a Variant containing an array whose elements are assigned values from the *argumentList*. The *argumentList* variable is a comma-delimited list of values that are assigned to consecutive elements of the array. Omitting *argumentList* creates an array with no elements. To use it,

you must redimension it with the ReDim command. The following statements create an array with the names of the days of the week:

```
Dim WeekDays
WeekDays = Array("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
    "Saturday", "Sunday")
FirstDay = WeekDays(0)           ' Monday
SecondDay = WeekDays(1)         ' Tuesday
```

The lower bound of an array created using the Array function is always 0. Notice also that the Array function *does not dimension the array*. It only assigns values to its elements.

LBound(arrayname[, dimension]) This function returns the smallest subscript for the indicated dimension of an array. The *arrayName* variable is the name of the array, and *dimension* is an integer indicating the dimension whose lower bound will be returned. If *dimension* is omitted, the first dimension is assumed. The LBound() function is used with the UBound() function, which returns the largest subscript of a dimension of a given array, to determine the size of the array.

UBound(arrayname[, dimension]) This function returns the largest subscript for the indicated dimension of an array. The *arrayName* variable is the name of the array, and *dimension* is an integer indicating the dimension whose upper bound will be returned. If *dimension* is omitted, the first dimension is assumed.

To scan all the elements of a one-dimensional array, use both the LBound() and the UBound() functions. The following statements convert the elements of the string array Strings() to uppercase:

```
Lower = LBound(Strings)
Upper = UBound(Strings)
For i = Lower to Upper
    Strings(i) = Ucase(Strings(i))
Next
```

IsArray(variable) This function returns True if its argument is an array. If the variable *Names* has been defined as:

```
Dim Strings(100)
```

the function:

```
IsArray(Strings)
```

returns True.

IsDate(expression) This function returns True if *expression* is a valid date. Use the IsDate() function to validate user data. Dates can be specified in various formats,

and validating them without the help of the `IsDate()` function would be a task on its own.

```
Bdate = InputBox("Please enter your birth date")
If IsDate(BDate) Then
    MsgBox "Date accepted"
End If
```

IsEmpty(variable) This function returns True if the *variable* is empty. An empty variable hasn't been initialized or explicitly set to Empty. After the execution of the following statements]:

```
numVar = 0
stringVar = ""
```

the variables *numVar* and *stringVar* are not empty because they have been initialized. If a variable has been declared with a Dim statement but not initialized or otherwise used by your program, it's empty. This variable must be initialized before it can be used; you can find out its status with the `IsEmpty()` function.

IsNull(expression) This function returns True if *expression* is Null. A Null value is a nonvalid value and is different from an Empty value. Regular variables can't be Null unless you assign the Null value to them. Object variables can be Null.

IsNumeric(expression) This function returns True if *expression* is a valid number. Use this function to check the validity of strings containing numeric data as follows:

```
age = InputBox("Please enter your age")
If Not IsNumeric(age) Then
    MsgBox("Please try again, this time with a valid number")
End If
```

IsObject(expression) This function returns a Boolean (True/False) value indicating whether *expression* represents an object variable. To find out the type of object, use the `TypeName()` or `VarType()` functions, which are described next.

TypeName(variable_name) This function returns a string that identifies the variable's type. It's similar to the `VarType()` function, only instead of returning an integer, it returns the name of the variable's type. The variable whose type you're examining with the `TypeName` function may have been declared implicitly or explicitly. Suppose you declare the following variables

```
Dim name As String
Dim a
```

The following statements produce the results shown (you can issue the statements in the Debug window and watch the values they return in the same window):

```
Print TypeName(name)
String
Print TypeName(a)
Empty
a = "I'm a string"
Print TypeName(a)
String
a = #5/11/97#
Print TypeName(a)
Date
Print TypeName(b)
Empty
```

Notice that the variable *b*, which wasn't declared, is Empty, but not Null. You must set a variable to Null from within your code.

Variable Type Conversion Functions

These functions convert their numeric argument to the corresponding type. With the introduction of the Variant data type, these functions are of little use. You can use them to document your code and show that the result of an operation should be of the particular type, but keep in mind that all operands in an arithmetic operation are first converted to double precision numbers for the greatest possible accuracy. Table A.8 lists the Variable Type Conversion functions and describes what they do.

TABLE A.8: The Variable Type Conversion Functions

FUNCTION	WHAT IT DOES
CBool(expression)	Converts its argument to Boolean (True/False) type; evaluates to True if expression evaluates to any nonzero value.
CByte(expression)	Converts its argument to Byte type.
CCur(expression)	Converts its argument to Currency type.
CDate(expression)	Converts its argument to Date type.
CDec(expression)	Converts its argument to Decimal type.
CDBl(expression)	Converts its argument to Double type.

Continued on next page

TABLE A.8 CONTINUED: The Variable Type Conversion Functions

FUNCTION	WHAT IT DOES
CInt(expression)	Converts its argument to Integer type.
CLng(expression)	Converts its argument to Long type.
CSng(expression)	Converts its argument to Single type.
CStr(expression)	Converts its argument to String type.
CVar(expression)	Converts its argument to Variant type. Numeric expressions are converted to doubles, and alphanumeric expressions are converted to strings.

CVErr() This function accepts as an argument a numeric value (which is an error number) and returns a variant of Error type containing the specified error number. The CVErr() function does not generate a runtime error. It can be the return value of a function, which may return a result (if no error occurred), or it can be an error object if something went wrong during its execution. See Chapter 3 for more information on using this function and for some examples.

String-Handling Functions

VBScript supports all the string-handling functions of Visual Basic. A typical script, similar to a Visual Basic application, spends much of its execution time manipulating strings (validity tests, parsing), and VBScript provides numerous functions for that purpose.

Asc(character), AscB(string), AscW(string) The Asc() function returns the character code corresponding to the character argument, and it works on all systems, regardless of whether they support Unicode characters.

The AscB() function is similar, except that instead of returning the character code for the first character, it returns the first byte.

The AscW() function returns the Unicode character code except on platforms that do not support Unicode, in which case, the behavior is identical to that of the Asc() function.

If you call the Asc() function with a string instead of a character, the character code of the string's first character is returned.

Chr(number), ChrB(number), ChrW(number) The Chr() function is the inverse of the Asc() function and returns the character associated with the specified

character code. Use this function to print characters that don't appear on the keyboard (such as line feeds or special symbols).

The `ChrB()` function is used with byte data contained in a string. Instead of returning a character, which may be one or two bytes, `ChrB()` always returns a single byte.

The `ChrW()` function returns a string containing the Unicode character except on platforms that don't support Unicode, in which case, the behavior is identical to that of the `Chr()` function.

LCase(string), UCase(string) The `LCase()` function accepts a string as an argument and converts it to lowercase; the `UCase()` function accepts a string as an argument and converts it to uppercase. After the following statements are executed:

```
Title = "Mastering Visual Basic"
LTitle = LCase(Title)
UTitle = UCase(Title)
```

the variable *LTitle* contains the string "mastering visual basic", and the variable *UTitle* contains the string "MASTERING VISUAL BASIC".

A useful function, which is missing from VBScript, is one that converts a string to "lower caps." In other words, a function that converts all the characters in the string to lowercase and then converts the first character of each word to uppercase. Now that you've seen all the string manipulation functions, you can write a `LowerCaps()` function as follows:

```
Function LowerCaps(str As String) As String

    position = InStr(str, " ") ' Locate first space
    While position < Len(str) ' while there are spaces in the
        string
            newWord = Left$(str, position) ' extract word
            newWord = LCase(newWord) ' and convert its first character to upper
            case
            newStr = newStr & UCase$(Left$(newWord, 1)) & Mid$(newWord, 2)
            str = Right$(str, Len(str) - position) ' remove word from string
            position = InStr(str, " ")
        Wend
        newWord = str ' convert the last word in the string
        newStr = newStr & UCase$(Left$(newWord, 1)) & Mid$(newWord, 2)
    LowerCaps = newStr ' return string in Lower Caps

End Function
```

The `LowerCaps()` function uses the `Instr()` function to locate successive instances of the space character in the string. It then isolates the words between spaces, changes their first character to uppercase and the rest of the word to lowercase, and appends them to the `NewStr` string. When the function exits, its value is the original string formatted in lower caps. If you call the `LowerCaps()` function with the following argument:

```
CompanyName = "ABC industrial, inc."  
UCString = LowerCaps(CompanyName)
```

the `UCString` variable's value will be:

```
"ABC Industrial, Inc."
```

InStr([startPos,] string1, string2[, compare]) The `InStr()` function returns the position of `string2` within `string1`. The first argument, which is optional, determines where in `string1` the search begins. If the `startPos` argument is omitted, the search begins at the first character of `string1`. If you execute the following statements:

```
str1 = "The quick brown fox jumped over the lazy dog"  
str2 = "the"  
Pos = Instr(str1, str2)
```

the variable `Pos` will have the value 33. If you search for the string "he" by setting:

```
str2 = "he"
```

the `Pos` variable's value will be 2. If the search begins at the third character in the string, the first instance of the string "he" after the third character will be located:

```
Pos = Instr(3, str1, str2)
```

This time the `Pos` variable will be 34.

The search is by default case-sensitive. To locate "the", "The", or "THE" in the string, specify the last optional argument whose value is 0 (default) for a case-sensitive search and 1 for a case-insensitive search.

The following statement locates the first occurrence of "the" in the string, regardless of case:

```
str1 = "The quick brown fox jumped over the lazy dog"  
str2 = "the"  
Pos = Instr(1, str1, str2, 1)
```

The value of `Pos` will be 1. If you set the last argument to 0, the `Pos` variable becomes 33. If you want to use the last optional argument of the `Instr()` function, you must also specify the first argument.

The `InStrB()` function is used with byte data contained in a string. Instead of returning the character position of the first occurrence of one string within another, `InStrB()` returns the byte position.

InStrRev(string1, string2, start, compare) This function returns the position of one string within another (as does the `InStr()` function), but it starts from the end of the string. The *string1* argument is the string being searched, and the *string2* argument is the string being searched for. The other two arguments are optional. The *start* argument is the starting position for the search. If it is omitted, the search begins at the last character. The *compare* argument indicates the kind of comparison to be used in locating the substrings, and its values are explained in the Filter entry. If *compare* is omitted, a binary comparison is performed.

StrComp(string1, string2 [, compare]) This function compares two strings and returns a value indicating the result, according to Table A.9.

TABLE A.9: The Values That the `StrComp()` Function Returns

VALUE	DESCRIPTION
-1	<i>string1</i> is less than <i>string2</i>
0	<i>string2</i> is equal to <i>string2</i>
1	<i>string1</i> is greater than <i>string2</i>
Null	<i>string1</i> and/or <i>string2</i> is Null

The last argument of the `StrComp()` function determines whether the comparison will be case-sensitive. If *compare* is 0 (or omitted), the comparison is case-sensitive. If it's 1, the comparison is case-insensitive.

The following function:

```
StrComp("Sybex", "SYBEX")
```

returns 1 ("Sybex" is greater than "SYBEX", because the lowercase y character is after the uppercase Y in the ASCII sequence). The following function:

```
StrComp("Sybex", "SYBEX", 1)
```

returns 0.

Left(string, number), LeftB(string, number) This function returns a number of characters from the beginning of a string. It accepts two arguments: the string and the number of characters to extract. If the string *date1* starts with the month

name, the following `Left()` function can extract the month's abbreviation from the string, as follows:

```
date1 = "December 25, 1995"
MonthName = Left(date1, 3)
```

The value of the *MonthName* variable is "Dec".

Use the `LeftB()` function with byte data contained in a string. Instead of specifying the number of characters, the arguments specify numbers of bytes.

Right(string, number), RightB(srting, number) This function is similar to the `Left` function, except that it returns a number of characters from the end of a string. The following statement:

```
Yr = Right(date1, 4)
```

assigns to the *Yr* variable the value "1995".

Use the `RightB()` function with byte data contained in a string. Instead of specifying the number of characters, the arguments specify numbers of bytes.

Mid(string, start, [length]), MidB(string, start[, length]) The `Mid()` function returns a section of a string of *length* characters, starting at position *start*. The following function:

```
Mid("09 February, 1957", 4, 8)
```

extracts the name of the month from the specified string.

If you omit the *length* argument, the `Mid()` function returns all the characters from the starting position to the end of the string. If the specified *length* exceeds the number of characters in the string after the *start* position, the remaining string from the *start* location is returned.

Use the `MidB()` function with byte data contained in a string. Instead of specifying the number of characters, the arguments specify numbers of bytes.

Len(string), LenB(string) The `Len()` function returns the length of a string. After the following statements execute:

```
Name = InputBox("Enter your first Name")
NameLen = Len(Name)
```

the variable *NameLen* contains the length of the string entered by the user in the Input Box.

The Len() function is frequently used as a first test for invalid input, as in the following lines:

```
If Len(Name) = 0 Then
    MsgBox "NAME field can't be empty"
Else
    MsgBox "Thank you for registering with us"
EndIf
```

Use the LenB() function with byte data contained in a string. Instead of returning the number of characters in a string, LenB() returns the number of bytes used to represent that string.

LTrim(string), RTrim(string), Trim(string) These functions trim the spaces in front of, after, and on either side of a string. They are frequently used in validating user input, as in the following.

```
If EMail <> "" Then
    MsgBox "Applications without an e-mail address won't be processed"
End If
```

The preceding won't, however, catch a string that only has spaces. To detect empty strings, use the Trim() function instead:

```
If Trim(EMail) = "" Then
    MsgBox "Invalid Entry!"
End If
```

Space(number) This function returns a string consisting of the specified number of spaces. The *number* argument is the number of spaces you want in the string. This function is useful for formatting output and clearing data in fixed-length strings.

String(number, character) This function returns a string of *number* characters, all of which are *character*. The following function:

```
String(12, "*")
```

returns the string "*****". Use the String() function to create long patterns of special symbols.

StrConv(string, conversion) This function returns a string variable converted as specified by the conversion argument, whose values are shown in Table A.10.

TABLE A.10: The Values Returned by the StrConv() Function

CONSTANT	VALUE	DESCRIPTION
vbUpperCase	1	Converts the string to uppercase characters.
vbLowerCase	2	Converts the string to lowercase characters.
vbProperCase	3	Converts the first letter of every word in string to uppercase.
vbWide*	4*	Converts narrow (single-byte) characters in string to wide (double-byte) characters.
vbNarrow*	8*	Converts wide (double-byte) characters in string to narrow (single-byte) characters.
vbKatakana*	16*	Converts Hiragana characters in string to Katakana characters.
vbHiragana*	32*	Converts Katakana characters in string to Hiragana characters.
vbUnicode	64	Converts the string to Unicode using the default code page of the system.
VbFromUnicode	128	Converts the string from Unicode to the default code page of the system.

*Applies to Far East locales.

To perform multiple conversions, add the corresponding values. To convert a string to lowercase and to Unicode format, use a statement such as the following:

```
newString = StrConv(txt, vbLowerCase + vbUnicode)
```

StrReverse(string) This function reverses the character order of its argument. Its syntax is:

```
StrReverse(string1)
```

The *string1* argument is the string whose characters will be reversed.

Filter(InputStrings, Value, Include, Compare) This function returns a zero-based array containing part of a string array, based on specified filter criteria. The *InputStrings* argument is a one-dimensional array of the strings to be searched, and the *Value* argument is the string to search for. The last two arguments are optional, and they indicate whether the function should contain substrings that include or exclude the specified value. If True, the Filter() function returns the subset of the array that contains *Value* as a substring. If False, the Filter() function returns the subset of the array that does not contain *Value* as a substring. The *Compare* argument indicates the kind of string comparison to be used and it can be any of the values in Table A.11.

TABLE A.11: The Values of the *Compare* Argument

Value	What It Does
<code>vbBinaryCompare</code>	Performs a binary (case sensitive) comparison.
<code>VbTextCompare</code>	Performs a textual (case insensitive) comparison.
<code>VbDatabaseCompare</code>	Performs a comparison based on information contained in the database in which the comparison is to be performed.

The array returned by the `Filter()` function contains only enough elements to contain the number of matched items. To use the `Filter()` function, you must declare an array without specifying the number of elements. Let's say you have declare the `Names` array as follows:

```
Dim Names
Names = Array("Abe", "John", "John", "Ron", "Jimmy")
```

You can find out if the name stored in the variable `myName` is in the `Names` array by calling the `Filter()` function as follows:

```
b = Filter(a, myName)
```

If the name stored in the variable `myName` isn't part of the `Names` array, `b` is an array with no elements. The function:

```
UBbound(b)
```

will return -1. If the name stored in the variable `myName` is "Abe", the upper bound of the array `b` will be 0, and the element `b(0)` will be "Abe". If the value of the `myName` variable is "John", the upper bound of the `Names` array will be 1, and the elements `b(0)` and `b(1)` will have the value "John".

You can also create an array that contains all the elements in the original, except for a specific value. The array `b` created with the statement:

```
b = Filter(a, "Ron", False)
```

will have 4 elements, which are all the elements of the array `Names` except for "Ron".

Replace(expression, find, replacewith, start, count, compare) This function returns a string in which a specified substring has been replaced with another substring a specified number of times. The *expression* argument is a string containing the string to be replaced, on which the `Replace` function acts. The *find* argument is the substring to be replaced, and *replacewith* is the replacement string. The remaining arguments are optional. The *start* argument is the character position where the search begins. If it is omitted, the search starts at the first character. The *count* argument is the number of replacements to be performed.

If it is omitted, all possible replacements will take place. Finally, the *compare* argument specifies the kind of comparison to be performed. The values of the *compare* argument are described in the Filter entry.

Join(list, delimiter) This function returns a string created by joining a number of substrings contained in an array. The *list* argument is a one-dimensional array containing substrings to be joined, and the optional *delimiter* argument is a character used to separate the substrings in the returned string. If it is omitted, the space character (" ") is used. If *delimiter* is a zero-length string, all items in the list are concatenated with no delimiters.

Split(expression, delimiter, count, compare) This function is the counterpart of the Join() function. It returns a zero-based, one-dimensional array containing a specified number of substrings. The *expression* argument is a string that contains the original string that will be broken into substrings, and the optional *delimiter* argument is a character used to delimit the substrings. If *delimiter* is omitted, the space character (" ") is assumed to be the delimiter. If *delimiter* is a zero-length string, a single-element array containing the entire expression string is returned. The *count* argument is also optional, and it determines the number of substrings to be returned. If it's -1, all substrings are returned. The last argument, *compare*, is also optional and indicates the kind of comparison to use when evaluating substrings. Its valid values are described in the Filter entry.

Let's say you have declared a string variable with the following path name:

```
path = "c:\win\desktop\ActiveX\Examples\VBSCRIPT"
```

The Split() function can extract the path's components and assign them to the parts array, with this statement:

```
parts = Split("c:\win\desktop\ActiveX\Examples\VBSCRIPT", "\")
```

To display the parts of the path, set up a loop such as the following:

```
For i = 0 To ubound(parts)
    MsgBox parts(i)
Next
```

Formatting Functions

Up through Visual Basic 5, there was only one function for formatting numbers and dates, the Format() function. Visual Basic 6 features a number of new formatting functions, which are specific to the data type they apply to (numbers, dollar amounts, and dates).

Format(expression[, format[, firstdayofweek[, firstweekofyear]]])

This function returns a string containing an expression formatted according to

instructions contained in a format expression. The *expression* variable is the number, string, or date to be converted, and *format* is a string that tells Visual Basic how to format the value. The string “hh:mm:ss”, for example, displays the expression as a time string. The Format() function is used to prepare numbers, dates, and strings for display. If you attempt to display the following expression:

```
Print atn(1)/4
```

the number 3.14159265358979 is displayed. If this value must appear in a text control, chances are good that it will overflow the available space.

You can control the number of decimal digits to be displayed with the following call to the Format() function:

```
Print Format(atn(1)*4, "##.####")
```

This statement displays the result 3.1416. If you are doing financial calculations and the result turns out to be 13,454.332345201, it would best to display it as \$13,454.33, with a statement such as the following:

```
amount = 13454.332345201
Print Format(amount, "$###,###.##")
```

These statements display the value \$13,454.33, which is a proper dollar amount.

The *firstdayofweek* argument determines which is the week’s first day and can have one of the values in Table A.12.

TABLE A.12: The Values of the *firstdayofweek* Argument

CONSTANT	VALUE	DESCRIPTION
vbUseSystem	0	Use NLS API setting.
VbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Similarly, the *firstweekofyear* determines which is the first week of the year, and it can have one of the values in Table A.13.

TABLE A.13: The Values of the *firstweekofyear* Argument

CONSTANT	VALUE	DESCRIPTION
vbUseSystem	0	Uses NLS API setting.
vbFirstJan1	1	Starts with the week of January 1.
vbFirstFourDays	2	Starts with the week that has at least four days in the year.
vbFirstFullWeek	3	Starts with the first full week of the year.

The *firstdayof week* and *firstweekofyear* arguments are used only in formatting dates.

There are many formatting strings for all three types of variables: numeric, string, and date and time. Table A.14, Table A.15, and Table A.16 show them.

TABLE A.14: User-Defined Time and Date Formatting

CHARACTER	DESCRIPTION
:	Time separator. In some locales, other characters may be used to represent the time separator. The time separator separates hours, minutes, and seconds when time values are formatted.
/	Date separator. In some locales, other characters may be used to represent the date separator. The date separator separates the day, month, and year when date values are formatted.
c	Displays date as dddddd and the time as ttttt.
d	Displays day as a number (1–31).
dd	Displays day as a number with a leading zero (01–31).
ddd	Displays day as an abbreviation (Sun–Sat).
dddd	Displays day as a full name (Sunday–Saturday).
dddddd	Displays complete date (including day, month, and year), formatted according to the system's short date format setting. The default short date format is m/d/y.
dddddd	Displays complete date, formatted according to the long date setting recognized by the system. The default long date format is mmmm dd, yyyy.
w	Displays day of the week as a number (1 for Sunday through 7 for Saturday).
ww	Displays week of the year as a number (1–54).

Continued on next page

TABLE A.14 CONTINUED: User-Defined Time and Date Formatting

CHARACTER	DESCRIPTION
m	Displays month as a number (1–12). If m immediately follows h or hh, the minute rather than the month is displayed.
mm	Displays month as a number with a leading zero (01–12). If m immediately follows h or hh, the minute rather than the month is displayed.
mmm	Displays month as an abbreviation (Jan–Dec).
mmm	Displays month as a full month name (January–December).
q	Displays quarter of the year as a number (1–4).
y	Displays day of the year as a number (1–366).
yy	Displays year as a 2-digit number (00–99).
yyyy	Displays year as a 4-digit number (100–9999).
h	Displays hours as a number (0–23).
hh	Displays hours with leading zeros (00–23).
n	Displays minutes without leading zeros (0–59).
nn	Displays minutes with leading zeros (00–59).
s	Displays seconds without leading zeros (0–59).
ss	Displays seconds with leading zeros (00–59).
tttt	Displays complete time (including hour, minute, and second), formatted using the time separator defined by the time format of the system. The default time format is h:mm:ss.
AM/PM	Uses the 12-hour format and displays the indication AM/PM.
am/pm	Uses the 12-hour format and displays the indication am/pm.
A/P	Uses the 12-hour format and displays the indication A/P
a/p	Uses the 12-hour format and displays the indication a/p.
AMPM	Uses the 12-hour format and displays the AM/PM string literal as defined by the system. Use the Regional Settings program in the Control Panel to set this literal for your system.

TABLE A.15: User-Defined Number Formatting

CHARACTER	WHAT IT IS OR DOES	DESCRIPTION
None		Displays the number with no formatting.
0	Digit placeholder	Displays a digit or a zero. If the expression has a digit in the position where the 0 appears in the format string, display it; otherwise, display a zero in that position. If the number has fewer digits than there are zeros in the format expression, leading or trailing zeros are displayed. If the number has more digits to the right of the decimal separator than there are zeros to the right of the decimal separator in the format expression, round the number to as many decimal places as there are zeros. If the number has more digits to the left of the decimal separator than there are zeros to the left of the decimal separator in the format expression, display the extra digits without modification.
#	Digit placeholder	Displays a digit or nothing. If the expression has a digit in the position where the # appears in the format string, display it; otherwise, display nothing in that position. This symbol works like the 0 digit placeholder, except that leading and trailing zeros aren't displayed if the number has the same or fewer digits than there are # characters on either side of the decimal separator in the format expression.
.	Decimal placeholder	The decimal placeholder determines how many digits are displayed to the left and right of the decimal separator. If the format expression contains only number signs to the left of this symbol, numbers smaller than 1 begin with a decimal separator. To display a leading zero displayed with fractional numbers, use 0 as the first digit placeholder to the left of the decimal separator.
%	Percentage placeholder	The expression is multiplied by 100. The percent character (%) is inserted in the position where it appears in the format string.
,	Thousand separator	Separates thousands from hundreds within a number greater than 1000. Two adjacent thousand separators or a thousand separator immediately to the left of the decimal separator (whether or not a decimal is specified) means "scale the number by dividing it by 1000, rounding as needed." For example, you can use the format string "##0,," to represent 100 million as 100. Numbers smaller than 1 million are displayed as 0. Two adjacent thousand separators in any position other than immediately to the left of the decimal separator are treated simply as specifying the use of a thousand separator.
:	Time separator	Separates hours, minutes, and seconds when time values are formatted.

Continued on next page

TABLE A.15 CONTINUED: User-Defined Number Formatting

CHARACTER	WHAT IT IS OR DOES	DESCRIPTION
/	Date separator	Separates the day, month, and year when date values are formatted.
E+, e-, e+	Scientific format	If the format expression contains at least one digit placeholder (0 or #) to the right of E-, E+, e-, or e+, the number is displayed in scientific format, and E or e is inserted between the number and its exponent. The number of digit placeholders to the right determines the number of digits in the exponent. Use E- or e- to place a minus sign next to negative exponents. Use E+ or e+ to place a minus sign next to negative exponents and a plus sign next to positive exponents.
+ \$ (space)	Display a literal character	To display a character other than one of those listed, precede it with a backslash (\) or enclose it in double quotation marks (" ").
\	Displays the next character in the format string	To display a character that has special meaning as a literal character, precede it with a backslash (\). The backslash itself isn't displayed. Using a backslash is the same as enclosing the next character in double quotation marks. To display a backslash, use two backslashes (\\). Examples of characters that can't be displayed as literal characters are the date-formatting and time-formatting characters (a, c, d, h, m, n, p, q, s, t, w, y, / and :), the numeric-formatting characters (#, 0, %, E, e, comma, and period), and the string-formatting characters (@, &, <, >, and !).
"ABC"	Displays the string inside the double quotation marks (" ")	To include a string in format from within code, you must use Chr(34) to enclose the text (34 is the character code for a quotation mark ("")).

TABLE A.16: User-Defined String Formatting

CHARACTER	WHAT IT IS OR DOES	DESCRIPTION
@	Character placeholder	Displays a character or a space. If the string has a character in the position where the at symbol (@) appears in the format string, it is displayed. Otherwise, a space in that position is displayed. Placeholders are filled from right to left unless there is an exclamation point character (!) in the format string.
&	Character placeholder	If the string has a character in the position where the ampersand (&) appears, it is displayed. Otherwise, nothing is displayed. Placeholders are filled from right to left unless there is an exclamation point character (!) in the format string.

Continued on next page

TABLE A.16 CONTINUED: User-Defined String Formatting

CHARACTER	WHAT IT IS OR DOES	DESCRIPTION
<	Force lowercase	All characters are first converted to lowercase.
>	Force uppercase	All characters are first converted to uppercase.
!	Scans placeholders from left to right	The default order is to use placeholders from right to left.

FormatCurrency(Expression, NumDigitsAfterDecimal, IncludeLeading-Digit, UseParensForNegativeNumbers, GroupDigits) This function returns a numeric expression formatted as a currency value (dollar amount) using the currency symbol defined in Control Panel. All arguments are optional, except for the *Expression* argument, which is the number to be formatted as currency. *NumDigitsAfterDecimal* is a value indicating how many digits will appear to the right of the decimal point. The default value is -1, which indicates that the computer's regional settings must be used. *IncludeLeadingDigit* is a tristate constant that indicates whether a leading zero is displayed for fractional values. The *UseParensForNegativeNumbers* argument is also a tristate constant that indicates whether to place negative values within parentheses. The last argument, *Group-Digits*, is another tristate constant that indicates whether numbers are grouped using the group delimiter specified in the computer's regional settings.

NOTE

A tristate variable is one that has three possible values: True, False, and Use-Default. The last value uses the computer's regional settings. When one or more optional arguments are omitted, values for omitted arguments are provided by the computer's regional settings.

FormatDateTime(Date, NamedFormat) This function formats a date or time value. The *Date* argument is a date value that will be formatted, and the optional argument *NamedFormat* indicates the date/time format to be used. It can have the values shown in Table A.17.

TABLE A.17: The Values for the *NamedFormat* Argument

Value	What It Does
VbGeneralDate	Displays a date and/or time. If a date part is present, it is displayed as a short date. If a time part is present, it is displayed as a long time. If both parts are present, both parts are displayed.

Continued on next page

TABLE A.17 CONTINUED: The Values for the *NamedFormat* Argument

Value	What It Does
VbLongDate	Displays a date using the long date format, as specified in the client computer's regional settings
VbShortDate	Displays a date using the short date format, as specified in the client computer's regional settings
VbLongTime	Displays a time using the time format specified in the client computer's regional settings
VbShortTime	Displays a time using the 24-hour format

FormatNumber(Expression, NumDigitsAfterDecimal, IncludeLeading-Digit, UseParensForNegativeNumbers, GroupDigits) This function returns a numeric value formatted as a number. The arguments of the FormatNumber() function are identical to the arguments of the FormatCurrency() function.

FormatPercent(Expression, NumDigitsAfterDecimal, IncludeLeading-Digit, UseParensForNegativeNumbers, GroupDigits) This function returns an expression formatted as a percentage (multiplied by 100) with a trailing % character. Its syntax and arguments are identical to the FormatCurrency() and FormatNumber() functions.

Math Functions

The following functions perform math operations. Their arguments are double-precision values and so are their results.

Abs(expression) This function returns the absolute value of its argument. Both Abs(1.01) and Abs(-1.01) return the value 1.01.

Atn(expression) This function returns the arctangent of an angle. The value returned is in radians. To convert it to degrees, multiply by $180/\pi$, where π is 3.14159.... To calculate π with double precision, use the following statement:

```
Atn(1)*4
```

Cos(expression) This function returns the cosine of an angle. The value of angle must be expressed in radians. To convert it to degrees, multiply by $180/\pi$, where π is 3.14159.... To calculate π with double precision, use the following statement:

```
Atn(1)*4
```

Exp(expression) This function returns the base of the natural logarithms to a power. The *expression* variable is the power, and its value can be a noninteger,

positive or negative value. The Exp() function complements the operation of the Log() function and is also called *antilogarithm*.

Int(expression), Fix(expression) Both these functions accept a numeric argument and return an integer value. If *expression* is positive, both functions behave the same. If it's negative, the Int() function returns the first negative integer less than or equal to *expression*, and Fix returns the first negative integer greater than or equal to *expression*. For example, Int(-1.1) returns -2, and Fix(-1.1) returns -1.

The functions Int(1.8) and Fix(1.8) both return 1. If you want to get rid of the decimal part of a number and round it as well, use the following expression:

```
Int(value + 0.5)
```

The *value* argument is the number to be rounded. The following function:

```
Int(100.1 + 0.5)
```

returns 100, and the function:

```
Int(100.8 + 0.5)
```

returns 101. This technique works with negative numbers as well. The following function:

```
Int(-100.1 + 0.5)
```

returns -100, and the function:

```
Int(-100.8 + 0.5)
```

returns -101.

Round(expression[, numdecimalplaces]) This function returns a numeric expression rounded to a specified number of decimal places. The *numdecimalplaces* argument is optional and indicates how many places to the right of the decimal are included in the rounding. If it is omitted, an integer value is returned.

The expression Round (3.49) returns 3, and the expression Round(3.51) returns 4. Use this new function to avoid statements such as Int(value + 0.5), which was used with previous versions of Visual Basic to round *value* to an integer.

Log(expression) The Log() function returns the natural logarithm of a number. The *expression* variable must be a positive number. The following expression:

```
Log(Exp(N))
```

returns N, and so will this expression:

```
Exp(Log(N))
```

If you combine the logarithm with the antilogarithm, you end up with the same number.

The natural logarithm is the logarithm to the base *e*, which is approximately 2.718282. The precise value of *e* is given by the function `Exp(1)`. To calculate logarithms to other bases, divide the natural logarithm of the number by the natural logarithm of the base. The following statement calculates the logarithm of a number in base 10:

```
Log10 = Log(number) / Log(10)
```

Hex(expression), Oct(expression) These two functions accept a decimal numeric value as an argument and return the octal and hexadecimal representation of the number in a string. The function `Hex(47)` returns the value “2F”, and the function `Oct(47)` returns the value “57”. To specify a hexadecimal number, prefix it with `&H`. The equivalent notation for octal numbers is `&O`. Given the following definitions:

```
Dvalue = 199: Ovalue = &O77
```

the function `Oct(Dvalue)` returns the string “307”, and the function `Hex(Ovalue)` returns “3F”. To display the decimal value of 3F, use a statement such as the following:

```
MsgBox ("The number 3F in decimal is " & &H3F)
```

The actual value that will be displayed is 63.

Rnd([expression]) This function returns a pseudo-random number in the range 0 to 1. The optional argument is called *seed* and is used as a starting point in the calculations that generate the random number.

If the *seed* is negative, the `Rnd()` function always returns the same number. As strange as this behavior may sound, you may need this feature to create repeatable random numbers to test your code. If *seed* is positive (or omitted), the `Rnd()` function returns the next random number in the sequence. Finally, if *seed* is zero, the `Rnd()` function returns the most recently generated random number.

In most cases, you don’t need a random number between 0 and 1, but between two other integer values. A playing card’s value is an integer in the range 1 through 13. To simulate the throw of a dice, you need a number in the range 1 through 6. To generate a random number in the range *lower* to *upper*, in which both bounds are integer numbers, use the following statement:

```
randomNumber = Int((upper - lower + 1)*rnd() + lower);
```

The following statement displays a random number in the range 1 to 49.

```
Debug.Print Int(Rnd * 49 + 1)
```

If you repeat this statement six times, you will get the Lotto’s six lucky numbers.

NOTE

The sequence of random numbers produced by Visual Basic is always the same! Let's say you have an application that displays three random numbers. If you stop and rerun the application, the same three numbers will be displayed. This is not a bug. It's a feature of Visual Basic that allows you to debug applications that use random numbers (if the sequence were different, you wouldn't be able to re-create the problem). To change this default behavior, call the `Randomize` statement at the beginning of your code. This statement will initialize the random number generator based on the value of the computer's Timer, and the sequences of random numbers will be different every time you run the application.

Sgn(expression) This function returns an integer indicating the sign of its argument: 1 if the argument is greater than zero, 0 if the argument is 0, and -1 if the argument is less than zero.

Sin(expression) This function returns the sine of an angle, specified in radians. See the `Cos()` entry.

Sqr(expression) This function returns the square root of a positive number. If the argument number is negative, the `Sqr()` function causes a runtime error, because by definition the square root of a negative number is undefined. If your program uses the `Sqr()` function, you must include some error trapping code such as the following:

```
If var >= 0 Then
    sqVar = Sqr(var)
Else
    MsgBox "The result can't be calculated"
End If
```

Tan(expression) This function returns the tangent of an angle, which must be expressed in radians.

Val(string) This function returns the numeric value of a string made up of digits. The `Val()` function starts reading the string from the left and stops when it reaches a character that isn't part of a number. If the value of the variable *a* is:

```
a = "18:6.05"
```

the statement:

```
Debug.Print Val(a)
```

returns 18.

Date and Time Functions

The date and time functions report (or set) the system's date and time. Visual Basic understands many formats for the date. Besides the common formats such as 2/9/1999 and 5/25/1995, it recognizes month names. Dates such as "February 1999" and "May 25, 1996" are valid date expressions. See the `Format(string)` entry for more on date and time formats.

Timer() This function returns a single number representing the number of seconds elapsed since midnight. It is frequently used for timing purposes, as long as the desired accuracy is not less than a second. To time an operation that takes a while to complete, use a structure such as the following:

```
T1 = Timer
    {lengthy calculations}
Debug.print Int(Timer - T1)
```

The last statement displays the integer part of the difference, which is the number of seconds elapsed since the calculations started.

Date() This function returns the current date in month/day/year format, unless you specified the UK date format (day/month/year). The following statement:

```
MsgBox "The system date is " & Date()
```

display a date such as 9/22/1998 in a message box. To set the system date, use the following statement:

```
date = "01/01/97"
```

Time() This function returns the system's time in AM/PM format. The following statement:

```
MsgBox "The system time is " & Time()
```

displays a time such as 5:13:05 PM in a message box. To set the system time, use the following statement:

```
Time = "13:00.00"
```

Now() This function returns both the system date and time, in the same format as they are reported by the `Date()` and `Time()` functions. The following statement:

```
Debug.print Now()
```

displays a date/time combination such as 9/13/1998 09:23:10 PM in a message box. There's only one space between the date and the time.

The `Now()` function is equivalent to the following pair of functions:

```
Date() & " " & Time()
```

Day(date) This function returns the day number of the date specified by the argument. The *date* argument must be a valid date (such as the value of the Date() or the Now() function). If the following function were called on 12/01/95, it would have returned 1.

```
Day(Date())
```

The Day(Now()) function returns the same result.

Weekday(date, [firstdayofweek]) This function returns an integer in the range 1 through 7, representing the day of the week (1 for Sunday, 2 for Monday, and so on). The first argument, *date*, can be any valid date expression. The second argument, which is optional, specifies the first day of the week. Set it to 1 to start counting from Sunday (the default), or set it to 2 to start counting from Monday. The value 3 corresponds to Tuesday, the value 4 corresponds to Wednesday, and so on.

The following code segment displays the name of the day:

```
DayNames = Array("Sunday", "Monday", "Tuesday", "Wednesday", "Thurs-  
day", "Friday", "Saturday")  
dayname = "Today it is " & DayNames(Weekday(Now)-1)  
Debug.Print dayname
```

Notice that the code subtracts 1 from the weekday to account for the array being zero based.

Month(date) This function returns an integer in the range 1 through 12, representing the number of the month of the specified date. Month(Date) returns the current month number.

MonthName(month[, abbreviate]) This function returns the name of the month specified by the *month* argument (a numeric value, which is 1 for January, 2 for February, and so on). The optional *abbreviate* argument is a Boolean value that indicates if the month name is to be abbreviated. By default, month names are not abbreviated.

Year(date) This function returns an integer representing the year of the date passed to it as an argument. The following function:

```
Year(Now())
```

returns the current year.

Hour(time) This function returns an integer in the range 0 through 24 that represents the hour of the specified time. The following statements:

```
Debug.Print Now  
Debug.Print Hour(Now)
```

produce something such as:

```
2/27/98 11:32:43 AM
11
```

Minute(time) This function returns an integer in the range 0 through 60 that represents the minute of the specified time. The following statements:

```
Debug.Print Now
Debug.Print Minute(Now)
```

produce something such as:

```
2/27/98 11:57:13 AM
57
```

Second(time) This function returns an integer in the range 0 through 60 that represents the seconds of the specified time. The following statements:

```
Debug.Print Now
Debug.Print Second(Now)
```

produce something such as:

```
2/27/98 11:57:03 AM
3
```

DateSerial(year, month, day) This function accepts three numeric arguments that correspond to a year, a month, and a day value and returns the corresponding date. The following statement:

```
MsgBox DateSerial(1999, 10, 1)
```

displays the string "10/1/99" in a message box.

Although hardly a useful operation, the DateSerial function can handle arithmetic operations with dates. For example, you can find out the date of the 90th day of the year by calling DateSerial() with the following arguments:

```
DateSerial(1996, 1, 90)
```

(30/3/96, if you are curious). To find out the date 1000 days from now, call the DateSerial function as follows:

```
DateSerial(Year(Date), Month(Date), Day(Date)+1000)
```

You can also add (or subtract) a number of months to the *month* argument and a number of years to the *year* argument.

DateValue(date) This function returns a variant of type Date. This function is handy if you are doing financial calculations based on the number of days between two dates. The difference in the following statement:

```
MsgBox DateValue("12/25/1996") - DateValue("12/25/1993")
```

is the number of days between the two dates, which happens to be 1096 days. You can verify this result by adding 1096 days to the earlier date:

```
MsgBox DateValue("12/25/1993") + 1096
```

or subtracting 1096 days from the later date:

```
MsgBox DateValue("12/25/1996") - 1096
```

TimeSerial(hours, minutes, seconds) This function returns a time, as specified by the three arguments. The following function:

```
TimeSerial(4, 10, 55)
```

returns:

```
4:10:55 AM
```

The TimeSerial() function is frequently used to calculate relative times. The following call to TimeSerial() returns the time 2 hours, 15 minutes, and 32 seconds before 4:13:40 PM:

```
TimeSerial(16 - 2, 13 - 15, 40 - 32)
```

which is 2:02:08 PM.

TimeValue(time) This function returns a variant of type Time. Like the DateValue() function, it can be used in operations that involve time. If the variables *Time1* and *Time2* are defined as follows:

```
Time1 = "04.10.55"
Time2 = "18.50.00"
```

you can find out the hours, minutes, and seconds between the two times with the following statements:

```
Diff = TimeValue(Time2) - TimeValue(Time1)
HourDiff = Hour(Diff)
MinDiff = Minute(Diff)
SecDiff = Second(Diff)
```

In this example, the values returned will be:

```
HourDiff=14
MinDiff=25
SecDiff=05
```

DateAdd(interval, number, date) This function returns a date that corresponds to a date plus some interval. The *interval* variable is a time unit (days, hours, weeks, and so on), *number* is the number of intervals to be added to the initial date, and *date* is the initial date. If *number* is positive, the date returned by DateAdd is in the future. If it's negative, the date returned is in the past. The interval argument can take one of the values in Table A.18.

TABLE A.18: The Values for the *interval* Argument

VALUE	DESCRIPTION
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday
ww	Week
h	Hour
n	Minute
s	Second

To find out the date one month after January 31, 1996, use the following statement:

```
Print DateAdd("m", 1, "31-Jan-96")
```

The result is:

```
2/29/96
```

and not an invalid date such as February 31.

DateAdd() also takes into consideration leap years. The following statement:

```
Print DateAdd("m", 1, "31-Jan-96")
```

displays the date 2/29/96 in the Immediate window.

The DateAdd() function is similar to the DateSerial() function, but it takes into consideration the actual duration of a month. For DateSerial(), each month has 30 days. The following statements:

```
day1=#1/31/1996#
Print DateSerial(year(day1), month(day1)+1, day(day1))
```

result in:

```
3/2/96
```

which is a date in March, not February.

DateDiff(interval, date1, date2[, firstdayofweek[, firstweekofyear]])

This function is the counterpart of the DateAdd() function and returns the number of intervals between two dates. The *interval* argument is the interval of time you use to calculate the difference between the two dates (see Table A.18, earlier in this appendix, for valid values). The *date1* and *date2* arguments are dates to be used in the calculation, and *firstdayofweek* and *firstweekofyear* are optional arguments that specify the first day of the week and the first week of the year.

Table A.19 shows the valid values for the *firstdayofweek* argument, and Table A.20 shows the valid values for the *firstweekofyear* argument.

TABLE A.19: The Values for the *firstdayofweek* Argument

CONSTANT	VALUE	DESCRIPTION
vbUseSystem	0	Use the NLS API setting.
vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

TABLE A.20: The Values for the *firstweekofyear* Argument

CONSTANT	VALUE	DESCRIPTION
vbUseSystem	0	Use the NLS API setting.
vbFirstJan1	1	Start with the week in which January 1 occurs (default).
vbFirstFourDays	2	Start with the first week that has at least four days in the new year.
vbFirstFullWeek	3	Start with the first full week of the year.

You can use the DateDiff() function to find how many days, weeks, and even seconds are between two dates. The following statement displays the number of

days and minutes until the turn of the century (or the time elapsed after the turn of century, depending on when you execute it):

```
century=#01/01/2000 00:00.00#
Print DateDiff("n", now(), century)
```

If you place this code in a Timer's Timer event, you can update a text control every second or every minute with the countdown to the end of the century. If you were to use the DateValue() function, as in the following:

```
Print minute(DateValue("01/01/2000 00:00.00") - DateValue(now()))
```

the result is a number in the range 0 through 60. You would have to take into consideration the difference of years, months, days, hours, and minutes to calculate the correct value.

DatePart(interval, date[,firstdayofweek[, firstweekofyear]]) This function returns the specified part of a given date. The *interval* argument is the desired format in which the part of the date will be returned (see Table A.18, earlier in this appendix, for its values), and *date* is the part of the date you are seeking. The optional arguments *firstdayofweek* and *firstdayofmonth* are the same as for the DateDiff() function. The following Print statements produce the result shown below them:

```
day1=#03/23/1996 15:03.30#
Print DatePart("yyyy", day1)
1996
Print DatePart("q", day1)
1
Print DatePart("m", day1)
3
Print DatePart("d", day1)
23
Print DatePart("w", day1)
7
Print DatePart("ww", day1)
12
Print DatePart("h", day1)
15
Print DatePart("n", day1)
3
Print DatePart("s", day1)
30
```

Financial Functions

The following functions can be used to calculate the parameters of a loan or an investment. I will explain only the functions that return the basic parameters of a

loan (such as the monthly payment or the loan's duration). The more advanced financial functions are described in the Visual Basic online documentation.

IPmt(rate, per, nper, pv[, fv[, type]]) This function returns the interest payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate. The result is a Double value.

The *rate* argument is a Double value specifying the interest rate for the payment period. For example, if the loan's annual percentage rate (APR) is 10 percent, paid in monthly installments, the rate per period is $0.1/12$, or 0.0083.

The *per* argument is a Double value specifying the current payment period; *per* is a number in the range 1 through *nper*.

The *nper* argument is a Double value specifying the total number of payments. For example, if you make monthly payments on a five-year loan, *nper* is $5 * 12$ (or 60).

The *Pv* argument is a Double value specifying the principal or present value. The loan amount is the present value to the lender of the monthly payments.

The *fv* argument is a Variant specifying the future value or cash balance after the final payment. The future value of a loan is \$0 because that's its value after the final payment. If you want to accumulate \$10,000 in your savings account over 5 years, however, the future value is \$10,000. If the *fv* argument is omitted, 0 is assumed.

The *type* argument is a Variant specifying when payments are due. Use 0 if payments are due at the end of the payment period; use 1 if payments are due at the beginning of the period. If the *type* argument is omitted, 0 is assumed.

Suppose you borrow \$30,000 at an annual percentage rate of 11.5%, to be paid off in 3 years with payments at the end of each month. Here's how you can calculate the total interest, as well as the monthly interest:

```
PVal = 30000&
FVal = 0&
APR = 0.115 / 12
MPayments = 3 * 12
For Period = 1 To MPayments
    IPayment = IPmt(APR, Period, MPayments, -PVal, FVal, 1)
    Debug.Print IPayment
    TotInt = TotInt + IPayment
Next Period
Debug.Print "Total interest paid: " & TotInt
```

The interest portion of the first payment is \$287.11, and the interest portion of the last payment is less than \$10. The total interest is \$5,276.02.

PPmt(rate, per, nper, pv[, fv[, type]]) This function is similar to the IPmt() function except that it returns the principal payment for a given period of a loan based on periodic, fixed payments and a fixed interest rate. For a description of the function's arguments, see the IPmt entry.

The code for calculating the principal payment of the previous example is nearly the same as that for calculating the interest:

```
PVal = 30000&
FVal = 0&
APR = 0.115 / 12
MPayments = 3 * 12
For Period = 1 To MPayments
    PPayment = PPmt(APR, Period, MPayments, -PVal, FVal, 1)
    Debug.Print PPayment
    TotPrincipal = TotPrincipal + PPayment
Next Period
Debug.Print "Total principal paid: " & TotPrincipal
```

In this example, the payments increase with time (that's how the total payment remains fixed). The total amount will be equal to the loan's amount, of course, and the fixed payment is the sum of the interest payment (as returned by the IPmt() function) plus the principal payment (as returned by the PPmt() function).

Pmt(rate, nper, pv[, fv[, type]]) This function is a combination of the IPmt() and PPmt() functions. It returns the payment (including both principal and interest) for a loan based on periodic, fixed payments and a fixed interest rate. For a description of the function's arguments, see the IPmt entry. Notice that the Pmt() function doesn't require the *per* argument because all payments are equal.

The code for calculating the monthly payment is similar to the code examples in the IPmt() and PPmt() entries:

```
PVal = 30000&
FVal = 0&
APR = 0.115 / 12
MPayments = 3 * 12
For Period = 1 To MPayments
    MPayment = Pmt(APR, Period, MPayments, -PVal, FVal, 1)
    Debug.Print MPayment
    TotAmount = TotAmount + MPayment
Next Period
Debug.Print "Total amount paid: " & TotAmount
```

FV(rate, nper, pmt[, pv[, type]]) This function returns the future value of a loan based on periodic, fixed payments and a fixed interest rate. The arguments

of the FV() function are explained in the IPmt() entry, and the *pmt* argument is the payment made in each period.

Suppose you want to calculate the future value of an investment with an interest rate of 6.25%, 48 monthly payments of \$180, and a present value of \$12,000. Use the FV() function with the following arguments:

```
Payment = 180
APR = 6.25 / 100
TotPmts = 48
PVal = 12000
FVal = FV(APR / 12, TotPmts, -Payment, -PVal, PayType)
MsgBox "After " & TotPmts & " your savings will be worth $" & FVal
```

The actual result is close to \$25,000.

NPer(rate, pmt, pv[, fv[, type]]) This function returns the number of periods for a loan based on periodic, fixed payments and a fixed interest rate. For a description of the function's arguments, see the IPmt entry.

Suppose you borrow \$25,000 at 11.5%, and you can afford to pay \$450 per month. To figure out what this means to your financial state in the future, you would like to know how many years it will take you to pay off the loan. Here's how you can use the Nper() function to do so:

```
FVal = 0
PVal = 25000
APR = 0.115 / 12
Payment = 450
PayType = 0
TotPmts = NPer(APR, -Payment, PVal, FVal, PayType)
If Int(TotPmts) <> TotPmts Then TotPmts = Int(TotPmts) + 1
Debug.Print "The loan's duration will be: " & TotPmts & " months"
```

The actual duration of this loan is 80 months, which corresponds to nearly 6.5 years. If the payment is increased from \$450 to \$500, the loan's duration will drop to 69 months, and a monthly payment of \$550 will bring the loan's duration down to 60 months.

Rate(nper, pmt, pv[, fv[, type[, guess]]) You use this function to figure out the interest rate per payment period for a loan. Its arguments are the same as with the previous financial functions, except for the *guess* argument, which is the estimated interest rate. If you omit the *guess* argument, the value 0.1 (10%) is assumed.

Suppose you want to borrow \$10,000 and pay it off in 36 months with a monthly payment of \$350 or less. Here's how you can use the `Rate()` function to calculate the interest rate:

```
FVal = 0
PVal = 10000
Payment = 350
Payments = 36
PayType = 0
guess = 0.1
IRate = Rate(Payments, -Payment, PVal, FVal, PayType, guess)
Debug.Print "The desired interest rate is: " & Irate * 12 * 100 & "%"
```

The interest rate is approximately 15.7%.

Table A.21 lists and describes the remaining financial functions.

TABLE A.21: Additional Financial Functions

Function	What It Returns
PV	The present value of an investment
NPV	A Double specifying the net present value of an investment based on a series of periodic cash flows and a discount rate
IRR	The internal rate of return for an investment
MIRR	A Double specifying the modified internal rate of return for a series of periodic cash flows
DDB	A Double specifying the depreciation of an asset for a specific time period using the double-declining balance method or some other method you specify
SYD	A Double specifying the sum-of-years' digits depreciation of an asset for a specified period
SLN	A Double specifying the straight-line depreciation of an asset for a single period

Color Functions

The following two functions specify color values that can be used with Visual Basic's drawing methods or with the properties that set the color of various controls (`ForeColor`, `BackColor`). The `QBColor()` function is the simpler one and is commonly used with business applications.

QBColor(color) This function returns a Long Integer representing the RGB color code corresponding to the specified color number. The *color* argument is a number in the range 0 through 15. Each value returns a different color, as shown in Table A.22.

TABLE A.22: The Values for the *color* Argument

NUMBER	COLOR
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Yellow
7	White
8	Gray
9	Light Blue
10	Light Green
11	Light Cyan
12	Light Red
13	Light Magenta
14	Light Yellow
15	Bright White

Use the `QBColor()` function to specify colors if you want to address the needs of users with the least-capable graphics adapter (one that can’t display more than the basic 16 colors). Also use it for business applications that don’t require many colors.

RGB(*red*, *green*, *blue*) This function returns a Long Integer representing a color value. The *red*, *green*, and *blue* arguments are integer values in the range 0 through 255, representing the values of the three basic colors. Table A.23 lists some of the most common colors and their corresponding red, green, and blue components. The colors correspond to the eight corners of the RGB color cube.

TABLE A.23: Common Colors and Their Corresponding RGB Components

COLOR	RED	GREEN	BLUE
Black	0	0	0
Blue	0	0	255
Green	0	255	0
Cyan	0	255	255
Red	255	0	0
Magenta	255	0	255
Yellow	255	255	0
White	255	255	255

For a detailed discussion on how to specify colors with the help of the RGB cube, see Chapter 6.

The following statement:

```
Text1.BackColor = RGB(255, 0, 0)
```

assigns a pure red color to the background of the Text1 control.

Registry Functions

In Chapter 13, I discussed the API functions for manipulating the Registry. Because the operating system and all other operations rely heavily on the Registry, you should exercise extreme care when using the Registry. Visual Basic provides a few special functions for storing values in the Registry. These functions are safer than the API functions, and they access only a single branch of the Registry (in other words, you can't ruin the branch of another application by mistake).

SaveSetting [appname, section, key, setting] This is a statement that stores a new setting in the Registry or updates an existing one. It's the only statement discussed in this section. The *appname* argument is the name of the application (or project) that stores the information in the Registry. It doesn't have to be the actual name of the application; it can be any string you supply, as long as it's unique for your application. The *section* argument is the name of the Registry section in which the key setting will be saved. The *key* argument is the name of the key setting that will be saved. The last argument, *setting*, is the value of the key to be saved. If *setting* can't be saved for any reason, a runtime trappable error is generated.

The following statements store the keys “Left” and “Top” in the Startup section of the application’s branch in the Registry:

```
SaveSetting "MyApp", "Startup", "Top", Me.Top
SaveSetting "MyApp", "Startup", "Left", Me.Left
```

These values should be saved to the Registry when the application ends, and they should be read when it starts, to place the Form on the desktop.

GetSetting(appname, section, key, [default]) This function returns a key setting from an application’s branch in the Registry. The arguments *appname*, *section*, and *key* are the same as in the previous entry. The last argument, *default*, is optional and contains the value to return if no value for the specified key exists in the Registry.

To read the key values stored in the Registry by the statements in the SaveSettings() entry, use the following code segment:

```
Me.Top = GetSetting("MyApp", "Startup", "Top", 100)
Me.Left = GetSetting("MyApp", "Startup", "Left", 150)
```

Don’t omit the default values here because the Form may be sized oddly if these keys are missing.

GetAllSettings(appname, section) This function returns a list of keys and their respective values from an application’s entry in the Registry. The *appname* argument is the name of the application (or project) whose key settings are requested. The *section* argument is the name of the section whose key settings are requested. The GetAllSettings() function returns all the keys and settings in the specified section of the Registry in a two-dimensional array. The element (0,0) of the array contains the name of the first key, and the elements (0,1) contains the setting of this key. The next two elements (1,0) and (1,1) contain the key and setting of the second element and so on. To find out how many keys are stored in the specific section of the Registry, use the LBound() and UBound() functions.

The following statement retrieves all the keys in the Startup section for the MyApp application and stores them in the array AllSettings:

```
AllSettings = GetAllSettings("MyApp", "Startup")
```

You can then set up a loop that scans the array and displays the key and setting pairs:

```
For i = LBound(AllSettings, 1) To UBound(AllSettings, 1)
    Debug.Print AllSettings(i, 0) & " = " & AllSettings(i, 1)
Next
```


Miscellaneous Functions

This section describes the functions used in controlling program flow.

Choose(index, choice-1[, choice-2, ... [, choice-n]]) This function selects and returns a value from a list of arguments. The *index* argument is a numeric value between 1 and the number of available choices. The following arguments are the available options. The function will return the first option if *index* is 1, the second option if *index* is 2, and so on. You can use the Choose() function to translate single digits to strings. The function IntToString() returns the name of the digit passed as an argument:

```
Function IntToString(int As Integer) As String
    IntToString = Choose (i+1, "zero", "one", "two", "three", _
        "four", "five", "six", "seven", "eight", "nine")
End Sub
```

If *index* is less than one or larger than the number of options, the Choose function returns a Null value.

IIf(expression, truepart, falsepart) This function returns one of two parts, depending on the evaluation of *expression*. If the *expression* argument is True, the *truepart* argument is returned. If *expression* is not True, the *falsepart* argument is returned. The IIf() function is equivalent to the following If clause:

```
If expression Then
    result = truepart
Else
    result = falsepart
End If
```

In many situations, this logic significantly reduces the amount of code. The Min() and Max() functions, for instance, can be easily implemented with the IIf() function:

```
Min = IIf(a<b, a, b)
Max = IIf(a>b, a, b)
```

Switch(expression1, value1, expression2, value2,...) This function evaluates a list of expressions and returns a value associated with the first expression in the list that happens to be True. If none of the expressions is True, the function returns Null. The following statement selects the proper quadrant depending on the signs of the variables X and Y:

```
Quadrant = Switch(X>0 and Y>0, 1, X<0 and Y>0, 1, X<0 and Y<0, 3, X<0
and Y<0, 4)
```

If both *X* and *Y* are negative, the *Quadrant* variable is assigned the value 1. If *X* is negative and *Y* is positive, *Quadrant* becomes 2, and so on. If either *X* or *Y* is zero, none of the expressions are True, and *Quadrant* becomes Null.

Environ() This function returns the environment variables (operating system variables set with the SET command). To access an environment variable use a numeric index or the variable's name. If you access the environment variables with an index value, as in:

```
Print Environ(2)
```

you'll get a string that contains both the name of the environment variable and its value, such as:

```
TMP=C:\WINDOWS\TEMP
```

To retrieve only the value of the TMP environment variable, use the expression:

```
Print Environ("TMP")
```

and the function will return the value:

```
C:\WINDOWS\TEMP
```

If you specify a nonexistent environment variable name, a zero-length string ("") is returned.

Shell(pathname [,windowstyle]) This function starts another application and returns a value representing the program's task ID if successful; otherwise, it returns zero. The *pathname* argument is the full path name of the application to be started and any arguments it may expect. The optional argument *windowstyle* determines the style of the window in which the application will be executed, and it can have one of the values shown in Table A.24.

TABLE A.24: The Values of the *pathname* Argument

Value	Description
vbHide (0)	The window is hidden, and focus is passed to it.
vbNormalFocus (1)	The window has the focus and is restored to its original size and position.
vbMinimizedFocus (2)	The window is displayed as an icon that has the focus.
vbMaximizedFocus (3)	The window is maximized and has the focus.
vbNormalNoFocus (4)	The window is restored to its most recent size and position. The currently active window remains active.
vbMinimizedNoFocus (6)	The window is displayed as an icon. The currently active window remains active.

The `Shell` function runs other programs asynchronously. This means that a program started with `Shell` might not finish executing before the statements following the `Shell` function are executed.