Visual Basic for Applications Reference

Visual Studio 6.0

Environ Function

See Also Example Specifics

Returns the **String** associated with an operating system environment variable.

Syntax

Environ({***envstring*** | ***number***}**)**

The **Environ** function syntax has these named arguments:

Part	Description
envstring	Optional. String expression containing the name of an environment variable.
number	Optional. Numeric expression corresponding to the numeric order of the environment string in the environment-string table. The ***number*** argument can be any numeric expression, but is rounded to a whole number before it is evaluated.

Remarks

If ***envstring*** can't be found in the environment-string table, a zero-length string ("") is returned. Otherwise, **Environ** returns the text assigned to the specified ***envstring***; that is, the text following the equal sign (=) in the environment-string table for that environment variable.

If you specify ***number***, the string occupying that numeric position in the environment-string table is returned. In this case, **Environ** returns all of the text, including ***envstring***. If there is no environment string in the specified position, **Environ** returns a zero-length string.

Visual Basic for Applications Reference

Environ Function Example

This example uses the **Environ** function to supply the entry number and length of the PATH statement from the environment-string table.

```
Dim EnvString, Indx, Msg, PathLen   ' Declare variables.
Indx = 1   ' Initialize index to 1.
Do
   EnvString = Environ(Indx)   ' Get environment
            ' variable.
   If Left(EnvString, 5) = "PATH=" Then   ' Check PATH entry.
      PathLen = Len(Environ("PATH"))   ' Get length.
      Msg = "PATH entry = " & Indx & " and length = " & PathLen
      Exit Do
   Else
      Indx = Indx + 1   ' Not PATH entry,
   End If   ' so increment.
Loop Until EnvString = ""
If PathLen > 0 Then
   MsgBox Msg   ' Display message.
Else
   MsgBox "No PATH environment variable exists."
End If
```

Visual Basic for Applications Reference

Visual Studio 6.0

EOF Function

See Also Example Specifics

Returns an Integer containing the Boolean value **True** when the end of a file opened for **Random** or sequential **Input** has been reached.

Syntax

EOF(*filenumber***)**

The required *filenumber* argument is an **Integer** containing any valid file number.

Remarks

Use **EOF** to avoid the error generated by attempting to get input past the end of a file.

The **EOF** function returns **False** until the end of the file has been reached. With files opened for **Random** or **Binary** access, **EOF** returns **False** until the last executed **Get** statement is unable to read an entire record.

With files opened for **Binary** access, an attempt to read through the file using the **Input** function until **EOF** returns **True** generates an error. Use the **LOF** and **Loc** functions instead of **EOF** when reading binary files with **Input**, or use **Get** when using the **EOF** function. With files opened for **Output**, **EOF** always returns **True**.

Visual Basic for Applications Reference

EOF Function Example

This example uses the **EOF** function to detect the end of a file. This example assumes that `MYFILE` is a text file with a few lines of text.

```
Dim InputData
Open "MYFILE" For Input As #1   ' Open file for input.
Do While Not EOF(1)   ' Check for end of file.
   Line Input #1, InputData   ' Read line of data.
   Debug.Print InputData   ' Print to the Immediate window.
Loop
Close #1   ' Close file.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Error Function

See Also Example Specifics

Returns the error message that corresponds to a given error number.

Syntax

Error[**(***errornumber***)**]

The optional *errornumber* argument can be any valid error number. If *errornumber* is a valid error number, but is not defined, **Error** returns the string "Application-defined or object-defined error." If *errornumber* is not valid, an error occurs. If *errornumber* is omitted, the message corresponding to the most recent run-time error is returned. If no run-time error has occurred, or *errornumber* is 0, **Error** returns a zero-length string ("").

Remarks

Examine the property settings of the **Err** object to identify the most recent run-time error. The return value of the **Error** function corresponds to the **Description** property of the **Err** object.

Visual Basic for Applications Reference

Error Function Example

This example uses the **Error** function to print error messages that correspond to the specified error numbers.

```
Dim ErrorNumber
For ErrorNumber = 61 To 64   ' Loop through values 61 - 64.
   Debug.Print Error(ErrorNumber)   ' Print error to Immediate window.
Next ErrorNumber
```

Visual Basic for Applications Reference

Visual Studio 6.0

Exp Function

See Also Example Specifics

Returns a **Double** specifying *e* (the base of natural logarithms) raised to a power.

Syntax

Exp(*number***)**

The required *number* argument is a Double or any valid numeric expression.

Remarks

If the value of *number* exceeds 709.782712893, an error occurs. The constant *e* is approximately 2.718282.

Note The **Exp** function complements the action of the **Log** function and is sometimes referred to as the antilogarithm.

Visual Basic for Applications Reference

Exp Function Example

This example uses the **Exp** function to return *e* raised to a power.

```
Dim MyAngle, MyHSin
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Visual Basic for Applications Reference

Visual Studio 6.0

FileAttr Function

See Also Example Specifics

Returns a Long representing the file mode for files opened using the **Open** statement.

Syntax

FileAttr(*filenumber*, *returntype*)

The **FileAttr** function syntax has these named arguments:

Part	Description
filenumber	Required; Integer. Any valid file number.
returntype	Required; **Integer**. Number indicating the type of information to return. Specify 1 to return a value indicating the file mode. On 16-bit systems only, specify 2 to retrieve an operating system file handle. ***Returntype*** 2 is not supported in 32-bit systems and causes an error.

Return Values

When the ***returntype*** argument is 1, the following return values indicate the file access mode:

Mode	Value
Input	1
Output	2
Random	4
Append	8
Binary	32

Visual Basic for Applications Reference

FileAttr Function Example

This example uses the **FileAttr** function to return the file mode and file handle of an open file. The file handle is returned only on 16-bit systems; on 32-bit systems, passing 2 as a second argument generates an error.

```
Dim FileNum, Mode, Handle
FileNum = 1   ' Assign file number.
Open "TESTFILE" For Append As FileNum   ' Open file.
Mode = FileAttr(FileNum, 1)   ' Returns 8 (Append file mode).
Handle = FileAttr(FileNum, 2)   ' Returns file handle.
Close FileNum   ' Close file.
```

Visual Basic for Applications Reference

Visual Studio 6.0

FileDateTime Function

See Also Example Specifics

Returns a **Variant** (**Date**) that indicates the date and time when a file was created or last modified.

Syntax

FileDateTime(*pathname***)**

The required *pathname* argument is a string expression that specifies a file name. The *pathname* may include the directory or folder, and the drive.

Visual Basic for Applications Reference

FileDateTime Function Example

This example uses the **FileDateTime** function to determine the date and time a file was created or last modified. The format of the date and time displayed is based on the locale settings of your system.

```
Dim MyStamp
' Assume TESTFILE was last modified on February 12, 1993 at 4:35:47 PM.
' Assume English/U.S. locale settings.
MyStamp = FileDateTime("TESTFILE")   ' Returns "2/12/93 4:35:47 PM".
```

Visual Basic for Applications Reference

Visual Studio 6.0

FileLen Function

See Also Example Specifics

Returns a Long specifying the length of a file in bytes.

Syntax

FileLen(*pathname***)**

The required *pathname* argument is a string expression that specifies a file. The *pathname* may include the directory or folder, and the drive.

Remarks

If the specified file is open when the **FileLen** function is called, the value returned represents the size of the file immediately before it was opened.

Note To obtain the length of an open file, use the **LOF** function.

Visual Basic for Applications Reference

FileLen Function Example

This example uses the **FileLen** function to return the length of a file in bytes. For purposes of this example, assume that `TESTFILE` is a file containing some data.

```
Dim MySize
MySize = FileLen("TESTFILE")   ' Returns file length (bytes).
```

Visual Basic for Applications Reference

Visual Studio 6.0

Filter Function

See Also Example Specifics

Description

Returns a zero-based array containing subset of a string array based on a specified filter criteria.

Syntax

Filter(*sourcesrray, match*[, *include*[, *compare*]])

The **Filter** function syntax has these named argument:

Part	Description
sourcearray	Required. One-dimensional array of strings to be searched.
match	Required. String to search for.
include	Optional. **Boolean** value indicating whether to return substrings that include or exclude ***match***. If ***include*** is **True**, **Filter** returns the subset of the array that contains ***match*** as a substring. If ***include*** is **False**, **Filter** returns the subset of the array that does not contain ***match*** as a substring.
compare	Optional. Numeric value indicating the kind of string comparison to use. See Settings section for values.

Settings

The ***compare*** argument can have the following values:

Constant	Value	Description
vbUseCompareOption	1	Performs a comparison using the setting of the **Option Compare** statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	Microsoft Access only. Performs a comparison based on information in your database.

Remarks

If no matches of ***match*** are found within ***sourcearray***, **Filter** returns an empty array. An error occurs if ***sourcearray*** is **Null** or is not a one-dimensional array.

The array returned by the **Filter** function contains only enough elements to contain the number of matched items.

Visual Basic for Applications Reference

Visual Studio 6.0

Format Function

See Also Example Specifics

Returns a **Variant** (**String**) containing an expression formatted according to instructions contained in a format expression.

Syntax

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

The **Format** function syntax has these parts:

Part	Description
expression	Required. Any valid expression.
format	Optional. A valid named or user-defined format expression.
firstdayofweek	Optional. A constant that specifies the first day of the week.
firstweekofyear	Optional. A constant that specifies the first week of the year.

Settings

The *firstdayofweek* argument has these settings:

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

The *firstweekofyear* argument has these settings:

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

Remarks

To Format	Do This
Numbers	Use predefined named numeric formats or create user-defined numeric formats.
Dates and times	Use predefined named date/time formats or create user-defined date/time formats.
Date and time serial numbers	Use date and time formats or numeric formats.
Strings	Create your own user-defined string formats.

If you try to format a number without specifying *format*, **Format** provides functionality similar to the **Str** function, although it is internationally aware. However, positive numbers formatted as strings using **Format** dont include a leading space reserved for the sign of the value; those converted using **Str** retain the leading space.

If you are formatting a non-localized numeric string, you should use a user-defined numeric format to ensure that you get the look you want.

Note If the **Calendar** property setting is Gregorian and *format* specifies date formatting, the supplied *expression* must be Gregorian. If the Visual Basic **Calendar** property setting is Hijri, the supplied *expression* must be Hijri.

If the calendar is Gregorian, the meaning of *format* expression symbols is unchanged. If the calendar is Hijri, all date format symbols (for example, *dddd*, *mmmm*, *yyyy*) have the same meaning but apply to the Hijri calendar. Format symbols remain in English; symbols that result in text display (for example, AM and PM) display the string (English or Arabic) associated with that symbol. The range of certain symbols changes when the calendar is Hijri.

Symbol	Range
d	1-30
dd	1-30

ww	1-51
mmm	Displays full month names (Hijri month names have no abbreviations).
y	1-355
yyyy	100-9666

Visual Basic for Applications Reference

Format Function Example

This example shows various uses of the **Format** function to format values using both named formats and user-defined formats. For the date separator (**/**), time separator (**:**), and AM/ PM literal, the actual formatted output displayed by your system depends on the locale settings on which the code is running. When times and dates are displayed in the development environment, the short time format and short date format of the code locale are used. When displayed by running code, the short time format and short date format of the system locale are used, which may differ from the code locale. For this example, English/U.S. is assumed.

`MyTime` and `MyDate` are displayed in the development environment using current system short time setting and short date setting.

```
Dim MyTime, MyDate, MyStr
MyTime = #17:04:23#
MyDate = #January 27, 1993#

' Returns current system time in the system-defined long time format.
MyStr = Format(Time, "Long Time")

' Returns current system date in the system-defined long date format.
MyStr = Format(Date, "Long Date")

MyStr = Format(MyTime, "h:m:s")   ' Returns "17:4:23".
MyStr = Format(MyTime, "hh:mm:ss AMPM")   ' Returns "05:04:23 PM".
MyStr = Format(MyDate, "dddd, mmm d yyyy")   ' Returns "Wednesday,
   ' Jan 27 1993".
' If format is not supplied, a string is returned.
MyStr = Format(23)   ' Returns "23".

' User-defined formats.
MyStr = Format(5459.4, "##,##0.00")   ' Returns "5,459.40".
MyStr = Format(334.9, "###0.00")   ' Returns "334.90".
MyStr = Format(5, "0.00%")   ' Returns "500.00%".
MyStr = Format("HELLO", "<")   ' Returns "hello".
MyStr = Format("This is it", ">")   ' Returns "THIS IS IT".
```

Visual Basic for Applications Reference

Visual Studio 6.0

FormatCurrency Function

See Also Example Specifics

Description

Returns an expression formatted as a currency value using the currency symbol defined in the system control panel.

Syntax

FormatCurrency(*Expression*[**,***NumDigitsAfterDecimal* [**,***IncludeLeadingDigit* [**,***UseParensForNegativeNumbers* [**,***GroupDigits*]]]]**)**

The **FormatCurrency** function syntax has these parts:

Part	Description
Expression	Required. Expression to be formatted.
NumDigitsAfterDecimal	Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is 1, which indicates that the computer's regional settings are used.
IncludeLeadingDigit	Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.
UseParensForNegativeNumbers	Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.
GroupDigits	Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the computer's regional settings. See Settings section for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have the following settings:

Constant	Value	Description
vbTrue	1	True
vbFalse	0	False

vbUseDefault	2	Use the setting from the computer's regional settings.

Remarks

When one or more optional arguments are omitted, the values for omitted arguments are provided by the computer's regional settings.

The position of the currency symbol relative to the currency value is determined by the system's regional settings.

Note All settings information comes from the **Regional Settings Currency** tab, except leading zero which comes from the **Number** tab.

Visual Basic for Applications Reference

Visual Studio 6.0

FormatDateTime Function

See Also Example Specifics

Description

Returns an expression formatted as a date or time.

Syntax

FormatDateTime(*Date*[**,***NamedFormat*]**)**

The **FormatDateTime** function syntax has these parts:

Part	Description
Date	Required. Date expression to be formatted.
NamedFormat	Optional. Numeric value that indicates the date/time format used. If omitted, **vbGeneralDate** is used.

Settings

The *NamedFormat* argument has the following settings:

Constant	Value	Description
vbGeneralDate	0	Display a date and/or time. If there is a date part, display it as a short date. If there is a time part, display it as a long time. If present, both parts are displayed.
vbLongDate	1	Display a date using the long date format specified in your computer's regional settings.
vbShortDate	2	Display a date using the short date format specified in your computer's regional settings.
vbLongTime	3	Display a time using the time format specified in your computer's regional settings.
vbShortTime	4	Display a time using the 24-hour format (hh:mm).

Visual Basic for Applications Reference

Visual Studio 6.0

FormatNumber Function

See Also Example Specifics

Description

Returns an expression formatted as a number.

Syntax

FormatNumber(*Expression*[**,***NumDigitsAfterDecimal* [**,***IncludeLeadingDigit* [**,***UseParensForNegativeNumbers* [**,***GroupDigits*]]]]**)**

The **FormatNumber** function syntax has these parts:

Part	Description
Expression	Required. Expression to be formatted.
NumDigitsAfterDecimal	Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is 1, which indicates that the computer's regional settings are used.
IncludeLeadingDigit	Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.
UseParensForNegativeNumbers	Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.
GroupDigits	Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the computer's regional settings. See Settings section for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have the following settings:

Constant	Value	Description
vbTrue	1	True
vbFalse	0	False

vbUseDefault	2	Use the setting from the computer's regional settings.

Remarks

When one or more optional arguments are omitted, the values for omitted arguments are provided by the computer's regional settings.

Note All settings information comes from the **Regional Settings Number** tab.

Visual Basic for Applications Reference

Visual Studio 6.0

FormatPercent Function

See Also Example Specifics

Description

Returns an expression formatted as a percentage (multipled by 100) with a trailing % character.

Syntax

FormatPercent(*Expression*[**,***NumDigitsAfterDecimal* [**,***IncludeLeadingDigit* [**,***UseParensForNegativeNumbers* [**,***GroupDigits*]]]]**)**

The **FormatPercent** function syntax has these parts:

Part	Description
Expression	Required. Expression to be formatted.
NumDigitsAfterDecimal	Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is 1, which indicates that the computer's regional settings are used.
IncludeLeadingDigit	Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.
UseParensForNegativeNumbers	Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.
GroupDigits	Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the computer's regional settings. See Settings section for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have the following settings:

Constant	Value	Description
vbTrue	1	True
vbFalse	0	False

vbUseDefault	2	Use the setting from the computer's regional settings.

Remarks

When one or more optional arguments are omitted, the values for omitted arguments are provided by the computer's regional settings.

Note All settings information comes from the **Regional Settings Number** tab.

Visual Basic for Applications Reference

Visual Studio 6.0

FreeFile Function

See Also Example Specifics

Returns an Integer representing the next file number available for use by the **Open** statement.

Syntax

FreeFile[(*rangenumber*)]

The optional *rangenumber* argument is a Variant that specifies the range from which the next free file number is to be returned. Specify a 0 (default) to return a file number in the range 1 255, inclusive. Specify a 1 to return a file number in the range 256 511.

Remarks

Use **FreeFile** to supply a file number that is not already in use.

Visual Basic for Applications Reference

FreeFile Function Example

This example uses the **FreeFile** function to return the next available file number. Five files are opened for output within the loop, and some sample data is written to each.

```
Dim MyIndex, FileNumber
For MyIndex = 1 To 5   ' Loop 5 times.
   FileNumber = FreeFile   ' Get unused file
      ' number.
   Open "TEST" & MyIndex For Output As #FileNumber   ' Create file name.
   Write #FileNumber, "This is a sample."   ' Output text.
   Close #FileNumber   ' Close file.
Next MyIndex
```

Visual Basic for Applications Reference

Visual Studio 6.0

FV Function

See Also Example Specifics

Returns a Double specifying the future value of an annuity based on periodic, fixed payments and a fixed interest rate.

Syntax

FV(*rate*, *nper*, *pmt*[, *pv*[, *type*]])

The **FV** function has these named arguments:

Part	Description
rate	Required. **Double** specifying interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
nper	Required. Integer specifying total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
pmt	Required. **Double** specifying payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
pv	Optional. Variant specifying present value (or lump sum) of a series of future payments. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make. If omitted, 0 is assumed.
type	Optional. **Variant** specifying when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

Remarks

An annuity is a series of fixed cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The ***rate*** and ***nper*** arguments must be calculated using payment periods expressed in the same units. For example, if ***rate*** is calculated using months, ***nper*** must also be calculated using months.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

Visual Basic for Applications Reference

FV Function Example

This example uses the **FV** function to return the future value of an investment given the percentage rate that accrues per period (`APR / 12`), the total number of payments (`TotPmts`), the payment (`Payment`), the current value of the investment (`PVal`), and a number that indicates whether the payment is made at the beginning or end of the payment period (`PayType`). Note that because `Payment` represents cash paid out, it's a negative number.

```
Dim Fmt, Payment, APR, TotPmts, PayType, PVal, FVal
Const ENDPERIOD = 0, BEGINPERIOD = 1   ' When payments are made.
Fmt = "###,###,##0.00"   ' Define money format.
Payment = InputBox("How much do you plan to save each month?")
APR = InputBox("Enter the expected interest annual percentage rate.")
If APR > 1 Then APR = APR / 100   ' Ensure proper form.
TotPmts = InputBox("For how many months do you expect to save?")
PayType = MsgBox("Do you make payments at the end of month?", vbYesNo)
If PayType = vbNo Then PayType = BEGINPERIOD Else PayType = ENDPERIOD
PVal = InputBox("How much is in this savings account now?")
FVal = FV(APR / 12, TotPmts, -Payment, -PVal, PayType)
MsgBox "Your savings will be worth " & Format(FVal, Fmt) & "."
```

Visual Basic for Applications Reference

Visual Studio 6.0

GetAllSettings Function

See Also Example Specifics

Returns a list of key settings and their respective values (originally created with **SaveSetting**) from an application's entry in the Windows registry.

Syntax

GetAllSettings(*appname, section*)

The **GetAllSettings** function syntax has these named arguments:

Part	Description
appname	Required. String expression containing the name of the application or project whose key settings are requested.
section	Required. String e**xpression** containing the name of the section whose key settings are requested. **GetAllSettings** returns a Variant whose contents is a two-dimensional array of strings containing all the key settings in the specified section and their corresponding values.

Remarks

GetAllSettings returns an uninitialized **Variant** if either ***appname*** or ***section*** does not exist.

Visual Basic for Applications Reference

GetAllSettings Function Example

This example first uses the **SaveSetting** statement to make entries in the Windows registry for the application specified as ***appname***, then uses the **GetAllSettings** function to display the settings. Note that application names and ***section*** names can't be retrieved with **GetAllSettings**. Finally, the **DeleteSetting** statement removes the application's entries.

```
' Variant to hold 2-dimensional array returned by GetAllSettings
' Integer to hold counter.
Dim MySettings As Variant, intSettings As Integer
' Place some settings in the registry.
SaveSetting appname := "MyApp", section := "Startup", _
key := "Top", setting := 75
SaveSetting "MyApp","Startup", "Left", 50
' Retrieve the settings.
MySettings = GetAllSettings(appname := "MyApp", section := "Startup")
   For intSettings = LBound(MySettings, 1) To UBound(MySettings, 1)
      Debug.Print MySettings(intSettings, 0), MySettings(intSettings, 1)
   Next intSettings
DeleteSetting "MyApp", "Startup"
```

Visual Basic for Applications Reference

Visual Studio 6.0

GetAttr Function

See Also Example Specifics

Returns an **Integer** representing the attributes of a file, directory, or folder.

Syntax

GetAttr(*pathname***)**

The required *pathname* argument is a string expression that specifies a file name. The *pathname* may include the directory or folder, and the drive.

Return Values

The value returned by **GetAttr** is the sum of the following attribute values:

Constant	Value	Description
vbNormal	0	Normal.
vbReadOnly	1	Read-only.
vbHidden	2	Hidden.
vbSystem	4	System file.
vbDirectory	16	Directory or folder.
vbArchive	32	File has changed since last backup.

Note These constants are specified by Visual Basic for Applications. The names can be used anywhere in your code in place of the actual values.

Remarks

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 the individual file attribute you want. If the result is not zero, that attribute is set for the named file. For example, the return value of the following **And** expression is zero if the Archive attribute is not set:

```
Result = GetAttr(FName) And vbArchive
```

A nonzero value is returned if the Archive attribute is set.

Visual Basic for Applications Reference

GetAttr Function Example

This example uses the **GetAttr** function to determine the attributes of a file and directory or folder.

```
Dim MyAttr
' Assume file TESTFILE has hidden attribute set.
MyAttr = GetAttr("TESTFILE")   ' Returns 2.

' Returns nonzero if hidden attribute is set on TESTFILE.
Debug.Print MyAttr And vbHidden

' Assume file TESTFILE has hidden and read-only attributes set.
MyAttr = GetAttr("TESTFILE")   ' Returns 3.

' Returns nonzero if hidden attribute is set on TESTFILE.
Debug.Print MyAttr And (vbHidden + vbReadOnly)

' Assume MYDIR is a directory or folder.
MyAttr = GetAttr("MYDIR")   ' Returns 16.
```

Visual Basic Reference

Visual Studio 6.0

GetAutoServerSettings Function

See Also Example

Returns information about the state of an ActiveX component's registration.

Syntax

object.**GetAutoServerSettings**([*progid*], [*clsid*])

The **GetAutoServerSettings** function syntax has these parts:

Part	Description
object	Required. An object expression that evaluates to an object in the Applies To list.
progid	Optional. A variant expression specifying the ProgID for the component.
clsid	Optional. A variant expression specifying the CLSID for the component.

Return Values

The **GetAutoServerSettings** function returns a Variant that contains an array of values about the given ActiveX component. The index values and descriptions are:

Value	Description
1	True if the ActiveX component is registered remotely.
2	Remote machine name.
3	RPC network protocol name.
4	RPC authentication level.

Remarks

If a value is missing or not available, the value will be an empty string. If there is an error during the method, then the return value will be a Variant of type Empty.

Visual Basic Reference

GetAutoServerSettings Function Example

This example retrieves information about a remotely registered object named "Hello":

```
Sub ViewHello()
   Dim oRegClass As New RegClass
   Dim vRC As Variant
   vRC = oRegClass.GetAutoServerSettings _
  ("HelloProj.HelloClass")
   If Not(IsEmpty(vRC)) Then
      If vRC(1) Then
         MsgBox "Hello is registered remotely on a " _
         & "server named: " & vRC(1)
      Else
         MsgBox "Hello is registered locally."
      End If
   End if
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

GetObject Function

See Also Example Specifics

Returns a reference to an object provided by an ActiveX component.

Syntax

GetObject([***pathname***] [**, *class***]**)**

The **GetObject** function syntax has these named arguments:

Part	Description
pathname	Optional; **Variant** (**String**). The full path and name of the file containing the object to retrieve. If ***pathname*** is omitted, ***class*** is required.
class	Optional; **Variant** (**String**). A string representing the class of the object.

The ***class*** argument uses the syntax *appname.objecttype* and has these parts:

Part	Description
appname	Required; **Variant** (**String**). The name of the application providing the object.
objecttype	Required; **Variant** (**String**). The type or class of object to create.

Remarks

Use the **GetObject** function to access an ActiveX object from a file and assign the object to an object variable. Use the **Set** statement to assign the object returned by **GetObject** to the object variable. For example:

```
Dim CADObject As Object
Set CADObject = GetObject("C:\CAD\SCHEMA.CAD")
```

When this code is executed, the application associated with the specified ***pathname*** is started and the object in the specified file is activated.

If ***pathname*** is a zero-length string (""), **GetObject** returns a new object instance of the specified type. If the ***pathname*** argument is omitted, **GetObject** returns a currently active object of the specified type. If no object of the specified type

exists, an error occurs.

Some applications allow you to activate part of a file. Add an exclamation point (**!**) to the end of the file name and follow it with a string that identifies the part of the file you want to activate. For information on how to create this string, see the documentation for the application that created the object.

For example, in a drawing application you might have multiple layers to a drawing stored in a file. You could use the following code to activate a layer within a drawing called `SCHEMA.CAD`:

```
Set LayerObject = GetObject("C:\CAD\SCHEMA.CAD!Layer3")
```

If you don't specify the object's ***class***, Automation determines the application to start and the object to activate, based on the file name you provide. Some files, however, may support more than one class of object. For example, a drawing might support three different types of objects: an **Application** object, a **Drawing** object, and a **Toolbar** object, all of which are part of the same file. To specify which object in a file you want to activate, use the optional ***class*** argument. For example:

```
Dim MyObject As Object
Set MyObject = GetObject("C:\DRAWINGS\SAMPLE.DRW", "FIGMENT.DRAWING")
```

In the example, `FIGMENT` is the name of a drawing application and `DRAWING` is one of the object types it supports.

Once an object is activated, you reference it in code using the object variable you defined. In the preceding example, you access properties and methods of the new object using the object variable `MyObject`. For example:

```
MyObject.Line 9, 90
MyObject.InsertText 9, 100, "Hello, world."
MyObject.SaveAs "C:\DRAWINGS\SAMPLE.DRW"
```

Note Use the **GetObject** function when there is a current instance of the object or if you want to create the object with a file already loaded. If there is no current instance, and you don't want the object started with a file loaded, use the **CreateObject** function.

If an object has registered itself as a single-instance object, only one instance of the object is created, no matter how many times **CreateObject** is executed. With a single-instance object, **GetObject** always returns the same instance when called with the zero-length string ("") syntax, and it causes an error if the ***pathname*** argument is omitted. You can't use **GetObject** to obtain a reference to a class created with Visual Basic.

Visual Basic for Applications Reference

GetObject Function Example

This example uses the **GetObject** function to get a reference to a specific Microsoft Excel worksheet (MyXL). It uses the worksheet's **Application** property to make Microsoft Excel visible, to close it, and so on. Using two API calls, the DetectExcel **Sub** procedure looks for Microsoft Excel, and if it is running, enters it in the Running Object Table. The first call to **GetObject** causes an error if Microsoft Excel isn't already running. In the example, the error causes the ExcelWasNotRunning flag to be set to True. The second call to **GetObject** specifies a file to open. If Microsoft Excel isn't already running, the second call starts it and returns a reference to the worksheet represented by the specified file, mytest.xls. The file must exist in the specified location; otherwise, the Visual Basic error Automation error is generated. Next the example code makes both Microsoft Excel and the window containing the specified worksheet visible. Finally, if there was no previous version of Microsoft Excel running, the code uses the **Application** object's **Quit** method to close Microsoft Excel. If the application was already running, no attempt is made to close it. The reference itself is released by setting it to **Nothing**.

```
' Declare necessary API routines:
Declare Function FindWindow Lib "user32" Alias _
"FindWindowA" (ByVal lpClassName as String, _
               ByVal lpWindowName As Long) As Long

Declare Function SendMessage Lib "user32" Alias _
"SendMessageA" (ByVal hWnd as Long,ByVal wMsg as Long, _
               ByVal wParam as Long, _
               ByVal lParam As Long) As Long

Sub GetExcel()
   Dim MyXL As Object   ' Variable to hold reference
                        ' to Microsoft Excel.
   Dim ExcelWasNotRunning As Boolean   ' Flag for final release.

' Test to see if there is a copy of Microsoft Excel already running.
   On Error Resume Next   ' Defer error trapping.
' Getobject function called without the first argument returns a
' reference to an instance of the application. If the application isn't
' running, an error occurs.
   Set MyXL = Getobject(, "Excel.Application")
   If Err.Number <> 0 Then ExcelWasNotRunning = True
   Err.Clear   ' Clear Err object in case error occurred.

' Check for Microsoft Excel. If Microsoft Excel is running,
' enter it into the Running Object table.
   DetectExcel

' Set the object variable to reference the file you want to see.
   Set MyXL = Getobject("c:\vb4\MYTEST.XLS")

' Show Microsoft Excel through its Application property. Then
' show the actual window containing the file using the Windows
' collection of the MyXL object reference.
   MyXL.Application.Visible = True
   MyXL.Parent.Windows(1).Visible = True
    Do manipulations of your  file here.
   ' ...
' If this copy of Microsoft Excel was not running when you
' started, close it using the Application property's Quit method.
' Note that when you try to quit Microsoft Excel, the
```

```
' title bar blinks and a message is displayed asking if you
' want to save any loaded files.
   If ExcelWasNotRunning = True Then
      MyXL.Application.Quit
   End IF

   Set MyXL = Nothing    ' Release reference to the
                         ' application and spreadsheet.
End Sub

Sub DetectExcel()
' Procedure dectects a running Excel and registers it.
   Const WM_USER = 1024
   Dim hWnd As Long
' If Excel is running this API call returns its handle.
   hWnd = FindWindow("XLMAIN", 0)
   If hWnd = 0 Then    ' 0 means Excel not running.
      Exit Sub
   Else
   ' Excel is running so use the SendMessage API
   ' function to enter it in the Running Object Table.
      SendMessage hWnd, WM_USER + 18, 0, 0
   End If
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

GetSetting Function

See Also Example Specifics

Returns a key setting value from an application's entry in the Windows registry.

Syntax

GetSetting(*appname*, *section*, *key*[, *default*])

The **GetSetting** function syntax has these named arguments:

Part	Description
appname	Required. String expression containing the name of the application or project whose key setting is requested.
section	Required. String expression containing the name of the section where the key setting is found.
key	Required. String expression containing the name of the key setting to return.
default	Optional. Expression containing the value to return if no value is set in the key setting. If omitted, ***default*** is assumed to be a zero-length string ("").

Remarks

If any of the items named in the **GetSetting** arguments do not exist, **GetSetting** returns the value of ***default***.

Visual Basic for Applications Reference

GetSetting Function Example

This example first uses the **SaveSetting** statement to make entries in the Windows registry (or .ini file on 16-bit Windows platforms) for the application specified as ***appname***, and then uses the **GetSetting** function to display one of the settings. Because the ***default*** argument is specified, some value is guaranteed to be returned. Note that ***section*** names can't be retrieved with **GetSetting**. Finally, the **DeleteSetting** statement removes all the application's entries.

```
' Variant to hold 2-dimensional array returned by GetSetting.
Dim MySettings As Variant
' Place some settings in the registry.
SaveSetting "MyApp","Startup", "Top", 75
SaveSetting "MyApp","Startup", "Left", 50

Debug.Print GetSetting(appname := "MyApp", section := "Startup", _
                       key := "Left", default := "25")

DeleteSetting "MyApp", "Startup"
```