Visual Basic for Applications Reference

Visual Studio 6.0

Abs Function

See Also Example Specifics

Returns a value of the same type that is passed to it specifying the absolute value of a number.

Syntax

Abs(*number***)**

The required *number* argument can be any valid numeric expression. If *number* contains Null, **Null** is returned; if it is an uninitialized variable, zero is returned.

Remarks

The absolute value of a number is its unsigned magnitude. For example, `ABS(-1)` and `ABS(1)` both return `1`.

Visual Basic for Applications Reference

Abs Function Example

This example uses the **Abs** function to compute the absolute value of a number.

```
Dim MyNumber
MyNumber = Abs(50.3)    ' Returns 50.3.
MyNumber = Abs(-50.3)    ' Returns 50.3.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Array Function

See Also Example Specifics

Returns a Variant containing an array.

Syntax

Array(*arglist***)**

The required *arglist* argument is a comma-delimited list of values that are assigned to the elements of the array contained within the **Variant**. If no arguments are specified, an array of zero length is created.

Remarks

The notation used to refer to an element of an array consists of the variable name followed by parentheses containing an index number indicating the desired element. In the following example, the first statement creates a variable named A as a **Variant**. The second statement assigns an array to variable A. The last statement assigns the value contained in the second array element to another variable.

```
Dim A As Variant
A = Array(10,20,30)
B = A(2)
```

The lower bound of an array created using the **Array** function is determined by the lower bound specified with the **Option Base** statement, unless **Array** is qualified with the name of the type library (for example **VBA.Array**). If qualified with the type-library name, **Array** is unaffected by **Option Base**.

Note A **Variant** that is not declared as an array can still contain an array. A **Variant** variable can contain an array of any type, except fixed-length strings and user-defined types. Although a **Variant** containing an array is conceptually different from an array whose elements are of type **Variant**, the array elements are accessed in the same way.

Visual Basic for Applications Reference

Array Function Example

This example uses the **Array** function to return a **Variant** containing an array.

```
Dim MyWeek, MyDay
MyWeek = Array("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
' Return values assume lower bound set to 1 (using Option Base
' statement).
MyDay = MyWeek(2)   ' MyDay contains "Tue".
MyDay = MyWeek(4)   ' MyDay contains "Thu".
```

Visual Basic for Applications Reference

Visual Studio 6.0

Asc Function

See Also Example Specifics

Returns an Integer representing the character code corresponding to the first letter in a string.

Syntax

Asc(*string***)**

The required *string* argument is any valid string expression. If the *string* contains no characters, a run-time error occurs.

Remarks

The range for returns is 0 255 on non-DBCS systems, but 32768 32767 on DBCS systems.

Note The **AscB** function is used with byte data contained in a string. Instead of returning the character code for the first character, **AscB** returns the first byte. The **AscW** function returns the Unicode character code except on platforms where Unicode is not supported, in which case, the behavior is identical to the **Asc** function.

Visual Basic for Applications Reference

Asc Function Example

This example uses the **Asc** function to return a character code corresponding to the first letter in the string.

```
Dim MyNumber
MyNumber = Asc("A")    ' Returns 65.
MyNumber = Asc("a")    ' Returns 97.
MyNumber = Asc("Apple")    ' Returns 65.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Atn Function

See Also Example Specifics

Returns a **Double** specifying the arctangent of a number.

Syntax

Atn(*number***)**

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

Remarks

The **Atn** function takes the ratio of two sides of a right triangle (*number*) and returns the corresponding angle in radians. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

The range of the result is -pi/2 to pi/2 radians.

To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

Note **Atn** is the inverse trigonometric function of **Tan**, which takes an angle as its argument and returns the ratio of two sides of a right triangle. Do not confuse **Atn** with the cotangent, which is the simple inverse of a tangent (1/tangent).

Visual Basic for Applications Reference

Atn Function Example

This example uses the **Atn** function to calculate the value of pi.

```
Dim pi
pi = 4 * Atn(1)   ' Calculate the value of pi.
```

Visual Basic for Applications Reference

Visual Studio 6.0

CallByName Function

See Also Example Specifics

Executes a method of an object, or sets or returns a property of an object.

Syntax

CallByName(*object*, *procname*, *calltype*,[*args()*])

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

Part	Description
object	Required; **Variant** (**Object**). The name of the object on which the function will be executed.
procname	Required; **Variant** (**String**). A string expression containing the name of a property or method of the object.
calltype	Required; **Constant**. A constant of type **vbCallType** representing the type of procedure being called.
args()	Optional: **Variant** (**Array**).

Remarks

The **CallByName** function is used to get or set a property, or invoke a method at run time using a string name.

In the following example, the first line uses **CallByName** to set the **MousePointer** property of a text box, the second line gets the value of the **MousePointe**r property, and the third line invokes the **Move** method to move the text box:

```
CallByName Text1, "MousePointer", vbLet, vbCrosshair
Result = CallByName (Text1, "MousePointer", vbGet)
CallByName Text1, "Move", vbMethod, 100, 100
```

Visual Basic for Applications Reference

Visual Studio 6.0

Choose Function

See Also Example Specifics

Selects and returns a value from a list of arguments.

Syntax

Choose(*index*, *choice-1*[, *choice-2*, ... [, *choice-n*]]**)**

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

Part	Description
index	Required. Numeric expression or field that results in a value between 1 and the number of available choices.
choice	Required. Variant expression containing one of the possible choices.

Remarks

Choose returns a value from the list of choices based on the value of *index*. If *index* is 1, **Choose** returns the first choice in the list; if *index* is 2, it returns the second choice, and so on.

You can use **Choose** to look up a value in a list of possibilities. For example, if *index* evaluates to 3 and *choice-1* = "one", *choice-2* = "two", and *choice-3* = "three", **Choose** returns "three". This capability is particularly useful if *index* represents the value in an option group.

Choose evaluates every choice in the list, even though it returns only one. For this reason, you should watch for undesirable side effects. For example, if you use the **MsgBox** function as part of an expression in all the choices, a message box will be displayed for each choice as it is evaluated, even though **Choose** returns the value of only one of them.

The **Choose** function returns a Null if *index* is less than 1 or greater than the number of choices listed.

If *index* is not a whole number, it is rounded to the nearest whole number before being evaluated.

Visual Basic for Applications Reference

Choose Function Example

This example uses the **Choose** function to display a name in response to an index passed into the procedure in the `Ind` parameter.

```
Function GetChoice(Ind As Integer)
   GetChoice = Choose(Ind, "Speedy", "United", "Federal")
End Function
```

Visual Basic for Applications Reference

Visual Studio 6.0

Chr Function

See Also Example Specifics

Returns a String containing the character associated with the specified character code.

Syntax

Chr(*charcode***)**

The required *charcode* argument is a Long that identifies a character.

Remarks

Numbers from 0 31 are the same as standard, nonprintable ASCII codes. For example, **Chr(**10**)** returns a linefeed character. The normal range for *charcode* is 0 255. However, on DBCS systems, the actual range for *charcode* is -32768 to 65535.

Note 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 where Unicode is not supported, in which case, the behavior is identical to the **Chr** function.

Visual Basic for Applications Reference

Chr Function Example

This example uses the **Chr** function to return the character associated with the specified character code.

```
Dim MyChar
MyChar = Chr(65)   ' Returns A.
MyChar = Chr(97)   ' Returns a.
MyChar = Chr(62)   ' Returns >.
MyChar = Chr(37)   ' Returns %.
```

This documentation is archived and is not being maintained.

Visual Basic for Applications Reference

Visual Studio 6.0

Command Function

See Also Example Specifics

Returns the argument portion of the command line used to launch Microsoft Visual Basic or an executable program developed with Visual Basic.

Syntax

Command

Remarks

When Visual Basic is launched from the command line, any portion of the command line that follows `/cmd` is passed to the program as the command-line argument. In the following example, `cmdlineargs` represents the argument information returned by the **Command** function.

```
VB /cmd cmdlineargs
```

For applications developed with Visual Basic and compiled to an .exe file, **Command** returns any arguments that appear after the name of the application on the command line. For example:

```
MyApp cmdlineargs
```

To find how command line arguments can be changed in the user interface of the application you're using, search Help for "command line arguments."

Visual Basic for Applications Reference

Command Function Example

This example uses the **Command** function to get the command line arguments in a function that returns them in a **Variant** containing an array.

```
Function GetCommandLine(Optional MaxArgs)
   'Declare variables.
   Dim C, CmdLine, CmdLnLen, InArg, I, NumArgs
   'See if MaxArgs was provided.
   If IsMissing(MaxArgs) Then MaxArgs = 10
   'Make array of the correct size.
   ReDim ArgArray(MaxArgs)
   NumArgs = 0: InArg = False
   'Get command line arguments.
   CmdLine = Command()
   CmdLnLen = Len(CmdLine)
   'Go thru command line one character
   'at a time.
   For I = 1 To CmdLnLen
      C = Mid(CmdLine, I, 1)
      'Test for space or tab.
      If (C <> " " And C <> vbTab) Then
         'Neither space nor tab.
         'Test if already in argument.
         If Not InArg Then
         'New argument begins.
         'Test for too many arguments.
            If NumArgs = MaxArgs Then Exit For
            NumArgs = NumArgs + 1
            InArg = True
         End If
         'Concatenate character to current argument.
         ArgArray(NumArgs) = ArgArray(NumArgs) & C
      Else
         'Found a space or tab.
         'Set InArg flag to False.
         InArg = False
      End If
   Next I
   'Resize array just enough to hold arguments.
   ReDim Preserve ArgArray(NumArgs)
   'Return Array in Function name.
   GetCommandLine = ArgArray()
End Function
```

Visual Basic for Applications Reference

Visual Studio 6.0

Cos Function

See Also Example Specifics

Returns a **Double** specifying the cosine of an angle.

Syntax

Cos(*number***)**

The required *number* argument is a Double or any valid numeric expression that expresses an angle in radians.

Remarks

The **Cos** function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.

The result lies in the range -1 to 1.

To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

Visual Basic for Applications Reference

Cos Function Example

This example uses the **Cos** function to return the cosine of an angle.

```
Dim MyAngle, MySecant
MyAngle = 1.3   ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
```

This documentation is archived and is not being maintained.

Visual Basic for Applications Reference

Visual Studio 6.0

CreateObject Function

See Also Example Specifics

Creates and returns a reference to an ActiveX object.

Syntax

CreateObject(*class,[servername]***)**

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

Part	**Description**
class	Required; **Variant** (**String**). The application name and class of the object to create.
servername	Optional; **Variant** (**String**). The name of the network server where the object will be created. If *servername* is an empty string (""), the local machine is used.

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

Every application that supports Automation provides at least one type of object. For example, a word processing application may provide an **Application** object, a **Document** object, and a **Toolbar** object.

To create an ActiveX object, assign the object returned by **CreateObject** to an object variable:

```
' Declare an object variable to hold the object
' reference. Dim as Object causes late binding.
Dim ExcelSheet As Object
Set ExcelSheet = CreateObject("Excel.Sheet")
```

This code starts the application creating the object, in this case, a Microsoft Excel spreadsheet. Once an object is created, you reference it in code using the object variable you defined. In the following example, you access properties and methods of the new object using the object variable, `ExcelSheet`, and other Microsoft Excel objects, including the `Application` object and the `Cells` collection.

```
' Make Excel visible through the Application object.
ExcelSheet.Application.Visible = True
' Place some text in the first cell of the sheet.
ExcelSheet.Application.Cells(1, 1).Value = "This is column A, row 1"
' Save the sheet to C:\test.xls directory.
ExcelSheet.SaveAs "C:\TEST.XLS"
' Close Excel with the Quit method on the Application object.
ExcelSheet.Application.Quit
' Release the object variable.
Set ExcelSheet = Nothing
```

Declaring an object variable with the `As Object` clause creates a variable that can contain a reference to any type of object. However, access to the object through that variable is late bound; that is, the binding occurs when your program is run. To create an object variable that results in early binding, that is, binding when the program is compiled, declare the object variable with a specific class ID. For example, you can declare and create the following Microsoft Excel references:

```
Dim xlApp As Excel.Application
Dim xlBook As Excel.Workbook
Dim xlSheet As Excel.WorkSheet
Set xlApp = CreateObject("Excel.Application")
Set xlBook = xlApp.Workbooks.Add
Set xlSheet = xlBook.Worksheets(1)
```

The reference through an early-bound variable can give better performance, but can only contain a reference to the class specified in the declaration.

You can pass an object returned by the **CreateObject** function to a function expecting an object as an argument. For example, the following code creates and passes a reference to a Excel.Application object:

```
Call MySub (CreateObject("Excel.Application"))
```

You can create an object on a remote networked computer by passing the name of the computer to the *servername* argument of **CreateObject**. That name is the same as the Machine Name portion of a share name: for a share named "\\MyServer\Public," *servername* is "MyServer."

Note Refer to COM documentation (see *Microsoft Developer Network*) for additional information on making an application visible on a remote networked computer. You may have to add a registry key for your application.

The following code returns the version number of an instance of Excel running on a remote computer named `MyServer`:

```
Dim xlApp As Object
Set xlApp = CreateObject("Excel.Application", "MyServer")
Debug.Print xlApp.Version
```

If the remote server doesnt exist or is unavailable, a run-time error occurs.

Note Use **CreateObject** when there is no current instance of the object. If an instance of the object is already running, a new instance is started, and an object of the specified type is created. To use the current instance, or to start the application and have it load a file, use the **GetObject** 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.

Visual Basic for Applications Reference

CreateObject Function Example

This example uses the **CreateObject** function to set a reference (`xlApp`) to Microsoft Excel. It uses the reference to access the **Visible** property of Microsoft Excel, and then uses the Microsoft Excel **Quit** method to close it. Finally, the reference itself is released.

```
Dim xlApp As Object    ' Declare variable to hold the reference.

Set xlApp = CreateObject("excel.application")
   ' You may have to set Visible property to True
   ' if you want to see the application.
xlApp.Visible = True
   ' Use xlApp to access Microsoft Excel's
   ' other objects.
xlApp.Quit   ' When you finish, use the Quit method to close
Set xlApp = Nothing    ' the application, then release the reference.
```

This documentation is archived and is not being maintained.

Visual Basic for Applications Reference

Visual Studio 6.0

CurDir Function

See Also Example Specifics

Returns a **Variant** (**String**) representing the current path.

Syntax

CurDir[(*drive*)]

The optional *drive* argument is a string expression that specifies an existing drive. If no drive is specified or if *drive* is a zero-length string (""), **CurDir** returns the path for the current drive.

Visual Basic for Applications Reference

CurDir Function Example

This example uses the **CurDir** function to return the current path.

```
' Assume current path on C drive is "C:\WINDOWS\SYSTEM" .
' Assume current path on D drive is "D:\EXCEL".
' Assume C is the current drive.
Dim MyPath
MyPath = CurDir   ' Returns "C:\WINDOWS\SYSTEM".
MyPath = CurDir("C")   ' Returns "C:\WINDOWS\SYSTEM".
MyPath = CurDir("D")   ' Returns "D:\EXCEL".
```

Visual Basic for Applications Reference

Visual Studio 6.0

CVErr Function

See Also Example Specifics

Returns a Variant of subtype **Error** containing an error number specified by the user.

Syntax

CVErr(*errornumber***)**

The required *errornumber* argument is any valid error number.

Remarks

Use the **CVErr** function to create user-defined errors in user-created procedures. For example, if you create a function that accepts several arguments and normally returns a string, you can have your function evaluate the input arguments to ensure they are within acceptable range. If they are not, it is likely your function will not return what you expect. In this event, **CVErr** allows you to return an error number that tells you what action to take.

Note that implicit conversion of an **Error** is not allowed. For example, you can't directly assign the return value of **CVErr** to a variable that is not a **Variant**. However, you can perform an explicit conversion (using **CInt**, **CDbl**, and so on) of the value returned by **CVErr** and assign that to a variable of the appropriate data type.

Visual Basic for Applications Reference

CVErr Function Example

This example uses the **CVErr** function to return a **Variant** whose **VarType** is **vbError** (10). The user-defined function `CalculateDouble` returns an error if the argument passed to it isn't a number. You can use **CVErr** to return user-defined errors from user-defined procedures or to defer handling of a run-time error. Use the **IsError** function to test if the value represents an error.

```
' Call CalculateDouble with an error-producing argument.
Sub Test()
   Debug.Print CalculateDouble("345.45robert")
End Sub
' Define CalculateDouble Function procedure.
Function CalculateDouble(Number)
   If IsNumeric(Number) Then
      CalculateDouble = Number * 2   ' Return result.
   Else
      CalculateDouble = CVErr(2001)   ' Return a user-defined error
   End If   ' number.
End Function
```

Visual Basic for Applications Reference

Visual Studio 6.0

Date Function

See Also Example Specifics

Returns a **Variant** (**Date**) containing the current system date.

Syntax

Date

Remarks

To set the system date, use the **Date** statement.

Date, and if the calendar is Gregorian, **Date$** behavior is unchanged by the **Calendar** property setting. If the calendar is Hijri, **Date$** returns a 10-character string of the form *mm-dd-yyyy*, where *mm* (01-12), *dd* (01-30) and *yyyy* (1400-1523) are the Hijri month, day and year. The equivalent Gregorian range is Jan 1, 1980 through Dec 31, 2099.

Visual Basic for Applications Reference

Date Function Example

This example uses the **Date** function to return the current system date.

```
Dim MyDate
MyDate = Date   ' MyDate contains the current system date.
```

Visual Basic for Applications Reference

Visual Studio 6.0

DateAdd Function

See Also Example Specifics

Returns a **Variant** (**Date**) containing a date to which a specified time interval has been added.

Syntax

DateAdd(*interval, number, date*)

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

Part	Description
interval	Required. String expression that is the interval of time you want to add.
number	Required. Numeric expression that is the number of intervals you want to add. It can be positive (to get dates in the future) or negative (to get dates in the past).
date	Required. **Variant** (**Date**) or literal representing date to which the interval is added.

Settings

The ***interval*** argument has these settings:

Setting	Description
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday
ww	Week

h	Hour
n	Minute
s	Second

Remarks

You can use the **DateAdd** function to add or subtract a specified time interval from a date. For example, you can use **DateAdd** to calculate a date 30 days from today or a time 45 minutes from now.

To add days to ***date***, you can use Day of Year ("y"), Day ("d"), or Weekday ("w").

The **DateAdd** function won't return an invalid date. The following example adds one month to January 31:

```
DateAdd("m", 1, "31-Jan-95")
```

In this case, **DateAdd** returns 28-Feb-95, not 31-Feb-95. If ***date*** is 31-Jan-96, it returns 29-Feb-96 because 1996 is a leap year.

If the calculated date would precede the year 100 (that is, you subtract more years than are in ***date***), an error occurs.

If ***number*** isn't a Long value, it is rounded to the nearest whole number before being evaluated.

Note The format of the return value for **DateAdd** is determined by **Control Panel** settings, not by the format that is passed in ***date*** argument.

Note For *date*, if the **Calendar** property setting is Gregorian, the supplied date must be Gregorian. If the calendar is Hijri, the supplied date must be Hijri. If month values are names, the name must be consistent with the current **Calendar** property setting. To minimize the possibility of month names conflicting with the current **Calendar** property setting, enter numeric month values (Short Date format).

Visual Basic for Applications Reference

DateAdd Function Example

This example takes a date and, using the **DateAdd** function, displays a corresponding date a specified number of months in the future.

```
Dim FirstDate As Date   ' Declare variables.
Dim IntervalType As String
Dim Number As Integer
Dim Msg
IntervalType = "m"   ' "m" specifies months as interval.
FirstDate = InputBox("Enter a date")
Number = InputBox("Enter number of months to add")
Msg = "New date: " & DateAdd(IntervalType, Number, FirstDate)
MsgBox Msg
```

Visual Basic for Applications Reference

Visual Studio 6.0

DateDiff Function

See Also Example Specifics

Returns a **Variant** (**Long**) specifying the number of time intervals between two specified dates.

Syntax

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

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

Part	Description
interval	Required. String expression that is the interval of time you use to calculate the difference between ***date1*** and ***date2***.
date1, ***date2***	Required; **Variant** (**Date**). Two dates you want to use in the calculation.
firstdayofweek	Optional. A constant that specifies the first day of the week. If not specified, Sunday is assumed.
firstweekofyear	Optional. A constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs.

Settings

The ***interval*** argument has these settings:

Setting	Description
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday

ww	Week
h	Hour
n	Minute
s	Second

The ***firstdayofweek*** argument has these settings:

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

Constant	Value	Description
vbUseSystem	0	Use the 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 new year.
vbFirstFullWeek	3	Start with first full week of the year.

Remarks

You can use the **DateDiff** function to determine how many specified time intervals exist between two dates. For example, you might use **DateDiff** to calculate the number of days between two dates, or the number of weeks between today and the end of the year.

To calculate the number of days between ***date1*** and ***date2***, you can use either Day of year ("y") or Day ("d"). When ***interval*** is Weekday ("w"), **DateDiff** returns the number of weeks between the two dates. If ***date1*** falls on a Monday, **DateDiff** counts the number of Mondays until ***date2***. It counts ***date2*** but not ***date1***. If ***interval*** is Week ("ww"), however, the **DateDiff** function returns the number of calendar weeks between the two dates. It counts the number of Sundays between ***date1*** and ***date2***. **DateDiff** counts ***date2*** if it falls on a Sunday; but it doesn't count ***date1***, even if it does fall on a Sunday.

If ***date1*** refers to a later point in time than ***date2***, the **DateDiff** function returns a negative number.

The ***firstdayofweek*** argument affects calculations that use the "w" and "ww" interval symbols.

If ***date1*** or ***date2*** is a date literal, the specified year becomes a permanent part of that date. However, if ***date1*** or ***date2*** is enclosed in double quotation marks (" "), and you omit the year, the current year is inserted in your code each time the ***date1*** or ***date2*** expression is evaluated. This makes it possible to write code that can be used in different years.

When comparing December 31 to January 1 of the immediately succeeding year, **DateDiff** for Year ("yyyy") returns 1 even though only a day has elapsed.

Note For ***date1*** and ***date2***, if the **Calendar** property setting is Gregorian, the supplied date must be Gregorian. If the calendar is Hijri, the supplied date must be Hijri.

Visual Basic for Applications Reference

DateDiff Function Example

This example uses the **DateDiff** function to display the number of days between a given date and today.

```
Dim TheDate As Date   ' Declare variables.
Dim Msg
TheDate = InputBox("Enter a date")
Msg = "Days from today: " & DateDiff("d", Now, TheDate)
MsgBox Msg
```

Visual Basic for Applications Reference

Visual Studio 6.0

DatePart Function

See Also Example Specifics

Returns a **Variant** (**Integer**) containing the specified part of a given date.

Syntax

DatePart(*interval, date*[*,firstdayofweek*[*, firstweekofyear*]])

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

Part	Description
interval	Required. String expression that is the interval of time you want to return.
date	Required. **Variant** (**Date**) value that you want to evaluate.
firstdayofweek	Optional. A constant that specifies the first day of the week. If not specified, Sunday is assumed.
firstweekofyear	Optional. A constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs.

Settings

The ***interval*** argument has these settings:

Setting	Description
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday

ww	Week
h	Hour
n	Minute
s	Second

The ***firstdayofweek*** argument has these settings:

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

The ***firstweekofyear*** argument has these settings:

Constant	Value	Description
vbUseSystem	0	Use the 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 new year.
vbFirstFullWeek	3	Start with first full week of the year.

Remarks

You can use the **DatePart** function to evaluate a date and return a specific interval of time. For example, you might use **DatePart** to calculate the day of the week or the current hour.

The ***firstdayofweek*** argument affects calculations that use the "w" and "ww" interval symbols.

If *date* is a date literal, the specified year becomes a permanent part of that date. However, if *date* is enclosed in double quotation marks (" "), and you omit the year, the current year is inserted in your code each time the *date* expression is evaluated. This makes it possible to write code that can be used in different years.

Note For *date*, if the **Calendar** property setting is Gregorian, the supplied date must be Gregorian. If the calendar is Hijri, the supplied date must be Hijri.

The returned date part is in the time period units of the current Arabic calendar. For example, if the current calendar is Hijri and the date part to be returned is the year, the year value is a Hijri year.

Visual Basic for Applications Reference

DatePart Function Example

This example takes a date and, using the **DatePart** function, displays the quarter of the year in which it occurs.

```
Dim TheDate As Date   ' Declare variables.
Dim Msg
TheDate = InputBox("Enter a date:")
Msg = "Quarter: " & DatePart("q", TheDate)
MsgBox Msg
```

Visual Basic for Applications Reference

Visual Studio 6.0

DateSerial Function

See Also Example Specifics

Returns a **Variant** (**Date**) for a specified year, month, and day.

Syntax

DateSerial(*year*, *month*, *day*)

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

Part	Description
year	Required; **Integer**. Number between 100 and 9999, inclusive, or a numeric expression.
month	Required; **Integer**. Any numeric expression.
day	Required; **Integer**. Any numeric expression.

Remarks

To specify a date, such as December 31, 1991, the range of numbers for each **DateSerial** argument should be in the accepted range for the unit; that is, 131 for days and 112 for months. However, you can also specify relative dates for each argument using any numeric expression that represents some number of days, months, or years before or after a certain date.

The following example uses numeric expressions instead of absolute date numbers. Here the **DateSerial** function returns a date that is the day before the first day (`1 - 1`), two months before August (`8 - 2`), 10 years before 1990 (`1990 - 10`); in other words, May 31, 1980.

```
DateSerial(1990 - 10, 8 - 2, 1 - 1)
```

Under Windows 98 or Windows 2000, two digit years for the ***year*** argument are interpreted based on user-defined machine settings. The default settings are that values between 0 and 29, inclusive, are interpreted as the years 20002029. The default values between 30 and 99 are interpreted as the years 19301999. For all other ***year*** arguments, use a four-digit year (for example, 1800).

Earlier versions of Windows interpret two-digit years based on the defaults described above. To be sure the function returns the proper value, use a four-digit year.

When any argument exceeds the accepted range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 35 days, it is evaluated as one month and some number of days, depending on where in the year it is applied. If any single argument is outside the range -32,768 to 32,767, an error occurs. If the date specified by the three arguments falls outside the acceptable range of dates, an error occurs.

Note For ***year***, ***month***, and ***day***, if the **Calendar** property setting is Gregorian, the supplied value is assumed to be Gregorian. If the **Calendar** property setting is Hijri, the supplied value is assumed to be Hijri.

The returned date part is in the time period units of the current Visual Basic calendar. For example, if the current calendar is Hijri and the date part to be returned is the year, the year value is a Hijri year. For the argument ***year***, values between 0 and 99, inclusive, are interpreted as the years 1400-1499. For all other ***year*** values, use the complete four-digit year (for example, 1520).

Visual Basic for Applications Reference

DateSerial Function Example

This example uses the **DateSerial** function to return the date for the specified year, month, and day.

```
Dim MyDate
' MyDate contains the date for February 12, 1969.
MyDate = DateSerial(1969, 2, 12)   ' Return a date.
```

Visual Basic for Applications Reference

Visual Studio 6.0

DateValue Function

See Also Example Specifics

Returns a **Variant** (**Date**).

Syntax

DateValue(*date***)**

The required *date* argument is normally a string expression representing a date from January 1, 100 through December 31, 9999. However, *date* can also be any expression that can represent a date, a time, or both a date and time, in that range.

Remarks

If *date* is a string that includes only numbers separated by valid date separators, **DateValue** recognizes the order for month, day, and year according to the Short Date format you specified for your system. **DateValue** also recognizes unambiguous dates that contain month names, either in long or abbreviated form. For example, in addition to recognizing 12/30/1991 and 12/30/91, **DateValue** also recognizes December 30, 1991 and Dec 30, 1991.

If the year part of *date* is omitted, **DateValue** uses the current year from your computer's system date.

If the *date* argument includes time information, **DateValue** doesn't return it. However, if *date* includes invalid time information (such as "89:98"), an error occurs.

Note For *date*, if the **Calendar** property setting is Gregorian, the supplied date must be Gregorian. If the calendar is Hijri, the supplied date must be Hijri. If the supplied date is Hijri, the argument *date* is a **String** representing a date from 1/1/100 (Gregorian Aug 2, 718) through 4/3/9666 (Gregorian Dec 31, 9999).

Visual Basic for Applications Reference

DateValue Function Example

This example uses the **DateValue** function to convert a string to a date. You can also use date literals to directly assign a date to a **Variant** or **Date** variable, for example, MyDate = #2/12/69#.

```
Dim MyDate
MyDate = DateValue("February 12, 1969")    ' Return a date.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Day Function

See Also Example Specifics

Returns a **Variant** (**Integer**) specifying a whole number between 1 and 31, inclusive, representing the day of the month.

Syntax

Day(*date***)**

The required *date* argument is any Variant, numeric expression, string expression, or any combination, that can represent a date. If *date* contains Null, **Null** is returned.

Note If the **Calendar** property setting is Gregorian, the returned integer represents the Gregorian day of the month for the date argument. If the calendar is Hijri, the returned integer represents the Hijri day of the month for the date argument.

Visual Basic for Applications Reference

Day Function Example

This example uses the **Day** function to obtain the day of the month from a specified date. In the development environment, the date literal is displayed in short format using the locale settings of your code.

```
Dim MyDate, MyDay
MyDate = #February 12, 1969#   ' Assign a date.
MyDay = Day(MyDate)   ' MyDay contains 12.
```

Visual Basic for Applications Reference

Visual Studio 6.0

DDB Function

See Also Example Specifics

Returns 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.

Syntax

DDB(*cost*, *salvage*, *life*, *period*[, *factor*])

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

Part	Description
cost	Required. **Double** specifying initial cost of the asset.
salvage	Required. **Double** specifying value of the asset at the end of its useful life.
life	Required. **Double** specifying length of useful life of the asset.
period	Required. **Double** specifying period for which asset depreciation is calculated.
factor	Optional. Variant specifying rate at which the balance declines. If omitted, 2 (double-declining method) is assumed.

Remarks

The double-declining balance method computes depreciation at an accelerated rate. Depreciation is highest in the first period and decreases in successive periods.

The ***life*** and ***period*** arguments must be expressed in the same units. For example, if ***life*** is given in months, ***period*** must also be given in months. All arguments must be positive numbers.

The **DDB** function uses the following formula to calculate depreciation for a given period:

Depreciation / ***period*** = ((***cost*** ***salvage***) * ***factor***) / ***life***

Visual Basic for Applications Reference

DDB Function Example

This example uses the **DDB** function to return the depreciation of an asset for a specified period given the initial cost (`InitCost`), the salvage value at the end of the asset's useful life (`SalvageVal`), the total life of the asset in years (`LifeTime`), and the period in years for which the depreciation is calculated (`Depr`).

```
Dim Fmt, InitCost, SalvageVal, MonthLife, LifeTime, DepYear, Depr
Const YRMOS = 12   ' Number of months in a year.
Fmt = "###,##0.00"
InitCost = InputBox("What's the initial cost of the asset?")
SalvageVal = InputBox("Enter the asset's value at end of its life.")
MonthLife = InputBox("What's the asset's useful life in months?")
Do While MonthLife < YRMOS   ' Ensure period is >= 1 year.
   MsgBox "Asset life must be a year or more."
   MonthLife = InputBox("What's the asset's useful life in months?")
Loop
LifeTime = MonthLife / YRMOS   ' Convert months to years.
If LifeTime <> Int(MonthLife / YRMOS) Then
   LifeTime = Int(LifeTime + 1)   ' Round up to nearest year.
End If
DepYear = CInt(InputBox("Enter year for depreciation calculation."))
Do While DepYear < 1 Or DepYear > LifeTime
   MsgBox "You must enter at least 1 but not more than " & LifeTime
   DepYear = InputBox("Enter year for depreciation calculation.")
Loop
Depr = DDB(InitCost, SalvageVal, LifeTime, DepYear)
MsgBox "The depreciation for year " & DepYear & " is " & _
Format(Depr, Fmt) & "."
```

Visual Basic for Applications Reference

Visual Studio 6.0

Dir Function

See Also Example Specifics

Returns a **String** representing the name of a file, directory, or folder that matches a specified pattern or file attribute, or the volume label of a drive.

Syntax

Dir[(*pathname*[, *attributes*])]

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

Part	Description
pathname	Optional. String expression that specifies a file name may include directory or folder, and drive. A zero-length string ("") is returned if *pathname* is not found.
attributes	Optional. Constant or numeric expression, whose sum specifies file attributes. If omitted, returns files that match *pathname* but have no attributes.

Settings

The *attributes* argument settings are:

Constant	Value	Description
vbNormal	0	(Default) Specifies files with no attributes.
vbReadOnly	1	Specifies read-only files in addition to files with no attributes.
vbHidden	2	Specifies hidden files in addition to files with no attributes.
VbSystem	4	Specifies system files in addition to files with no attributes.
vbVolume	8	Specifies volume label; if any other attributed is specified, **vbVolume** is ignored.
vbDirectory	16	Specifies directories or folders in addition to files with no attributes.

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

Remarks

Dir supports the use of multiple character (*****) and single character (**?**) wildcards to specify multiple files.

> **Security Note** Do not make decisions about the contents of a file based on the file name extension. For example, a file named Form1.vb may not be a Visual Basic source file.

Visual Basic for Applications Reference

Dir Function Example

This example uses the **Dir** function to check if certain files and directories exist.

```
Dim MyFile, MyPath, MyName
' Returns "WIN.INI"  if it exists.
MyFile = Dir("C:\WINDOWS\WIN.INI")

' Returns filename with specified extension. If more than one *.ini
' file exists, the first file found is returned.
MyFile = Dir("C:\WINDOWS\*.INI")

' Call Dir again without arguments to return the next *.INI file in the
' same directory.
MyFile = Dir

' Return first *.TXT file with a set hidden attribute.
MyFile = Dir("*.TXT", vbHidden)

' Display the names in C:\ that represent directories.
MyPath = "c:\"   ' Set the path.
MyName = Dir(MyPath, vbDirectory)   ' Retrieve the first entry.
Do While MyName <> ""   ' Start the loop.
   ' Ignore the current directory and the encompassing directory.
   If MyName <> "." And MyName <> ".." Then
      ' Use bitwise comparison to make sure MyName is a directory.
      If (GetAttr(MyPath & MyName) And vbDirectory) = vbDirectory Then
         Debug.Print MyName   ' Display entry only if it
      End If   ' it represents a directory.
   End If
   MyName = Dir   ' Get next entry.
Loop
```

Visual Basic for Applications Reference

Visual Studio 6.0

DoEvents Function

See Also Example Specifics

Yields execution so that the operating system can process other events.

Syntax

DoEvents( )

Remarks

The **DoEvents** function returns an Integer representing the number of open forms in stand-alone versions of Visual Basic, such as Visual Basic, Professional Edition. **DoEvents** returns zero in all other applications.

DoEvents passes control to the operating system. Control is returned after the operating system has finished processing the events in its queue and all keys in the **SendKeys** queue have been sent.

DoEvents is most useful for simple things like allowing a user to cancel a process after it has started, for example a search for a file. For long-running processes, yielding the processor is better accomplished by using a Timer or delegating the task to an ActiveX EXE component.. In the latter case, the task can continue completely independent of your application, and the operating system takes case of multitasking and time slicing.

Caution Any time you temporarily yield the processor within an event procedure, make sure the procedure is not executed again from a different part of your code before the first call returns; this could cause unpredictable results. In addition, do not use **DoEvents** if other applications could possibly interact with your procedure in unforeseen ways during the time you have yielded control.

Visual Basic for Applications Reference

DoEvents Function Example

This example uses the **DoEvents** function to cause execution to yield to the operating system once every 1000 iterations of the loop. **DoEvents** returns the number of open Visual Basic forms, but only when the host application is Visual Basic.

```
' Create a variable to hold number of Visual Basic forms loaded
' and visible.
Dim I, OpenForms
For I = 1 To 150000   ' Start loop.
   If I Mod 1000 = 0 Then    ' If loop has repeated 1000 times.
      OpenForms = DoEvents   ' Yield to operating system.
   End If
Next I   ' Increment loop counter.
```