Visual Basic for Applications Reference

Visual Studio 6.0

RaiseEvent Statement

See Also Example Specifics

Fires an event declared at module level within a class, form, or document.

Syntax

RaiseEvent *eventname* [**(***argumentlist***)**]

The required *eventname* is the name of an event declared within the module and follows Basic variable naming conventions.

The **RaiseEvent** statement syntax has these parts:

Part	Description
eventname	Required. Name of the event to fire.
argumentlist	Optional. Comma-delimited list of variables, arrays, or expressions The *argumentlist* must be enclosed by parentheses. If there are no arguments, the parentheses must be omitted.

Remarks

If the event has not been declared within the module in which it is raised, an error occurs. The following fragment illustrates an event declaration and a procedure in which the event is raised.

```
' Declare an event at module level of a class module
Event LogonCompleted (UserName as String)

Sub
   ' Raise the event.
   RaiseEvent LogonCompleted ("AntoineJan")
End Sub
```

If the event has no arguments, including empty parentheses, in the **RaiseEvent**, invocation of the event causes an error. You can't use **RaiseEvent** to fire events that are not explicitly declared in the module. For example, if a form has a Click event, you can't fire its Click event using **RaiseEvent**. If you declare a Click event in the form module, it shadows the forms own Click event. You can still invoke the forms Click event using normal syntax for calling the event, but not using the **RaiseEvent** statement.

Event firing is done in the order that the connections are established. Since events can have **ByRef** parameters, a process that connects late may receive parameters that have been changed by an earlier event handler.

Visual Basic for Applications Reference

RaiseEvent Statment Example

The following example uses events to count off seconds during a demonstration of the fastest 100 meter race. The code illustrates all of the event-related methods, properties, and statements, including the **RaiseEvent** statement.

The class that raises an event is the event source, and the classes that implement the event are the sinks. An event source can have multiple sinks for the events it generates. When the class raises the event, that event is fired on every class that has elected to sink events for that instance of the object.

The example also uses a form (`Form1`) with a button (`Command1`), a label (`Label1`), and two text boxes (`Text1` and `Text2`). When you click the button, the first text box displays "From Now" and the second starts to count seconds. When the full time (9.84 seconds) has elapsed, the first text box displays "Until Now" and the second displays "9.84"

The code for `Form1` specifies the initial and terminal states of the form. It also contains the code executed when events are raised.

```
Option Explicit

Private WithEvents mText As TimerState

Private Sub Command1_Click()
    Text1.Text = "From Now"
    Text1.Refresh
    Text2.Text = "0"
    Text2.Refresh
    Call mText.TimerTask(9.84)
End Sub

Private Sub Form_Load()
    Command1.Caption = "Click to Start Timer"
    Text1.Text = ""
    Text2.Text = ""
    Label1.Caption = "The fastest 100 meters ever run took this long:"
    Set mText = New TimerState
    End Sub

Private Sub mText_ChangeText()
    Text1.Text = "Until Now"
    Text2.Text = "9.84"
End Sub

Private Sub mText_UpdateTime(ByVal dblJump As Double)
    Text2.Text = Str(Format(dblJump, "0"))
    DoEvents
End Sub
```

The remaining code is in a class module named TimerState. Included among the commands in this module are the **Raise Event** statements.

```
Option Explicit
Public Event UpdateTime(ByVal dblJump As Double)
Public Event ChangeText()

Public Sub TimerTask(ByVal Duration As Double)
```

```
    Dim dblStart As Double
    Dim dblSecond As Double
    Dim dblSoFar As Double
    dblStart = Timer
    dblSoFar = dblStart

    Do While Timer < dblStart + Duration
        If Timer - dblSoFar >= 1 Then
            dblSoFar = dblSoFar + 1
            RaiseEvent UpdateTime(Timer - dblStart)
        End If
    Loop

    RaiseEvent ChangeText

End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

Randomize Statement

See Also Example Specifics

Initializes the random-number generator.

Syntax

Randomize [*number*]

The optional *number* argument is a Variant or any valid numeric expression.

Remarks

Randomize uses *number* to initialize the **Rnd** function's random-number generator, giving it a new seed value. If you omit *number*, the value returned by the system timer is used as the new seed value.

If **Randomize** is not used, the **Rnd** function (with no arguments) uses the same number as a seed the first time it is called, and thereafter uses the last generated number as a seed value.

Note To repeat sequences of random numbers, call **Rnd** with a negative argument immediately before using **Randomize** with a numeric argument. Using **Randomize** with the same value for *number* does not repeat the previous sequence.

> **Security Note** Because the **Random** statement and the **Rnd** function start with a seed value and generate numbers that fall within a finite range, the results may be predictable by someone who knows the algorithm used to generate them. Consequently, the **Random** statement and the **Rnd** function should not be used to generate random numbers for use in cryptography.

Visual Basic for Applications Reference

Randomize Statement Example

This example uses the **Randomize** statement to initialize the random-number generator. Because the number argument has been omitted, **Randomize** uses the return value from the **Timer** function as the new seed value.

```
Dim MyValue
Randomize   ' Initialize random-number generator.

MyValue = Int((6 * Rnd) + 1)   ' Generate random value between 1 and 6.
```

Visual Basic for Applications Reference

Visual Studio 6.0

ReDim Statement

See Also Example Specifics

Used at procedure level to reallocate storage space for dynamic array variables.

Syntax

ReDim [**Preserve**] *varname*(*subscripts*) [**As** *type*] [**,** *varname*(*subscripts*) [**As** *type*]] . . .

The **ReDim** statement syntax has these parts:

Part	Description
Preserve	Optional. Keyword used to preserve the data in an existing array when you change the size of the last dimension.
varname	Required. Name of the variable; follows standard variable naming conventions.
subscripts	Required. Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax: [*lower* **To**] *upper* [**,**[*lower* **To**] *upper*] . . . When not explicitly stated in *lower*, the lower bound of an array is controlled by the **Option Base** statement. The lower bound is zero if no **Option Base** statement is present.
type	Optional. Data type of the variable; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (for variable-length strings), **String** * *length* (for fixed-length strings), Object, Variant, a user-defined type, or an object type. Use a separate **As** *type* clause for each variable being defined. For a **Variant** containing an array, *type* describes the type of each element of the array, but doesn't change the **Variant** to some other type.

Remarks

The **ReDim** statement is used to size or resize a dynamic array that has already been formally declared using a **Private**, **Public**, or **Dim** statement with empty parentheses (without dimension subscripts).

You can use the **ReDim** statement repeatedly to change the number of elements and dimensions in an array. However, you can't declare an array of one data type and later use **ReDim** to change the array to another data type, unless the array is contained in a **Variant**. If the array is contained in a **Variant**, the type of the elements can be changed using an **As** *type* clause, unless youre using the **Preserve** keyword, in which case, no changes of data type are permitted.

If you use the **Preserve** keyword, you can resize only the last array dimension and you can't change the number of dimensions at all. For example, if your array has only one dimension, you can resize that dimension because it is the last and

only dimension. However, if your array has two or more dimensions, you can change the size of only the last dimension and still preserve the contents of the array. The following example shows how you can increase the size of the last dimension of a dynamic array without erasing any existing data contained in the array.

```
ReDim X(10, 10, 10)
. . .
ReDim Preserve X(10, 10, 15)
```

Similarly, when you use **Preserve**, you can change the size of the array only by changing the upper bound; changing the lower bound causes an error.

If you make an array smaller than it was, data in the eliminated elements will be lost. If you pass an array to a procedure by reference, you can't redimension the array within the procedure.

When variables are initialized, a numeric variable is initialized to 0, a variable-length string is initialized to a zero-length string (""), and a fixed-length string is filled with zeros. **Variant** variables are initialized to Empty. Each element of a user-defined type variable is initialized as if it were a separate variable. A variable that refers to an object must be assigned an existing object using the **Set** statement before it can be used. Until it is assigned an object, the declared object variable has the special value **Nothing**, which indicates that it doesn't refer to any particular instance of an object.

Caution The **ReDim** statement acts as a declarative statement if the variable it declares doesn't exist at module level or procedure level. If another variable with the same name is created later, even in a wider scope, **ReDim** will refer to the later variable and won't necessarily cause a compilation error, even if **Option Explicit** is in effect. To avoid such conflicts, **ReDim** should not be used as a declarative statement, but simply for redimensioning arrays.

Note To resize an array contained in a **Variant**, you must explicitly declare the **Variant** variable before attempting to resize its array.

Visual Basic for Applications Reference

ReDim Statement Example

This example uses the **ReDim** statement to allocate and reallocate storage space for dynamic-array variables. It assumes the **Option Base** is **1**.

```
Dim MyArray() As Integer   ' Declare dynamic array.
Redim MyArray(5)   ' Allocate 5 elements.
For I = 1 To 5   ' Loop 5 times.
   MyArray(I) = I   ' Initialize array.
Next I
```

The next statement resizes the array and erases the elements.

```
Redim MyArray(10)   ' Resize to 10 elements.
For I = 1 To 10   ' Loop 10 times.
   MyArray(I) = I   ' Initialize array.
Next I
```

The following statement resizes the array but does not erase elements.

```
Redim Preserve MyArray(15)   ' Resize to 15 elements.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Rem Statement

See Also Example Specifics

Used to include explanatory remarks in a program.

Syntax

Rem *comment*

You can also use the following syntax:

' *comment*

The optional *comment* argument is the text of any comment you want to include. A space is required between the **Rem** keyword and *comment*.

Remarks

If you use line numbers or line labels, you can branch from a **GoTo** or **GoSub** statement to a line containing a **Rem** statement. Execution continues with the first executable statement following the **Rem** statement. If the **Rem** keyword follows other statements on a line, it must be separated from the statements by a colon (**:**).

You can use an apostrophe (') instead of the **Rem** keyword. When you use an apostrophe, the colon is not required after other statements.

Visual Basic for Applications Reference

Rem Statement Example

This example illustrates the various forms of the **Rem** statement, which is used to include explanatory remarks in a program.

```
Dim MyStr1, MyStr2
MyStr1 = "Hello": Rem Comment after a statement separated by a colon.
MyStr2 = "Goodbye"    ' This is also a comment; no colon is needed.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Reset Statement

See Also Example Specifics

Closes all disk files opened using the **Open** statement.

Syntax

Reset

Remarks

The **Reset** statement closes all active files opened by the **Open** statement and writes the contents of all file buffers to disk.

Visual Basic for Applications Reference

Reset Statement Example

This example uses the **Reset** statement to close all open files and write the contents of all file buffers to disk. Note the use of the **Variant** variable `FileNumber` as both a string and a number.

```
Dim FileNumber
For FileNumber = 1 To 5   ' Loop 5 times.
   ' Open file for output. FileNumber is concatenated into the string
   ' TEST for the file name, but is a number following a #.
   Open "TEST" & FileNumber For Output As #FileNumber
   Write #FileNumber, "Hello World"   ' Write data to file.
Next FileNumber
Reset   ' Close files and write contents
      ' to disk.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Resume Statement

See Also Example Specifics

Resumes execution after an error-handling routine is finished.

Syntax

Resume [**0**]

Resume Next

Resume *line*

The **Resume** statement syntax can have any of the following forms:

Statement	Description
Resume	If the error occurred in the same procedure as the error handler, execution resumes with the statement that caused the error. If the error occurred in a called procedure, execution resumes at the statement that last called out of the procedure containing the error-handling routine.
Resume Next	If the error occurred in the same procedure as the error handler, execution resumes with the statement immediately following the statement that caused the error. If the error occurred in a called procedure, execution resumes with the statement immediately following the statement that last called out of the procedure containing the error-handling routine (or **On Error Resume Next** statement).
Resume *line*	Execution resumes at *line* specified in the required *line* argument. The *line* argument is a line label or line number and must be in the same procedure as the error handler.

Remarks

If you use a **Resume** statement anywhere except in an error-handling routine, an error occurs.

Visual Basic for Applications Reference

Resume Statement Example

This example uses the **Resume** statement to end error handling in a procedure, and then resume execution with the statement that caused the error. Error number 55 is generated to illustrate using the **Resume** statement.

```
Sub ResumeStatementDemo()
   On Error GoTo ErrorHandler   ' Enable error-handling routine.
   Open "TESTFILE" For Output As #1   ' Open file for output.
   Kill "TESTFILE"   ' Attempt to delete open file.
   Exit Sub   ' Exit Sub to avoid error handler.
ErrorHandler:   ' Error-handling routine.
   Select Case Err.Number   ' Evaluate error number.
      Case 55   ' "File already open" error.
         Close #1   ' Close open file.
      Case Else
         ' Handle other situations here....
   End Select
   Resume   ' Resume execution at same line
   ' that caused the error.
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

RmDir Statement

See Also Example Specifics

Removes an existing directory or folder.

Syntax

RmDir *path*

The required *path* argument is a string expression that identifies the directory or folder to be removed. The *path* may include the drive. If no drive is specified, **RmDir** removes the directory or folder on the current drive.

Remarks

An error occurs if you try to use **RmDir** on a directory or folder containing files. Use the **Kill** statement to delete all files before attempting to remove a directory or folder.

Visual Basic for Applications Reference

RmDir Statement Example

This example uses the **RmDir** statement to remove an existing directory or folder.

```
' Assume that MYDIR is an empty directory or folder.
RmDir "MYDIR"   ' Remove MYDIR.
```

Visual Basic for Applications Reference

Visual Studio 6.0

RSet Statement

See Also Example Specifics

Right aligns a string within a string variable.

Syntax

RSet *stringvar* = *string*

The **RSet** statement syntax has these parts:

Part	Description
stringvar	Required. Name of string variable.
string	Required. String expression to be right-aligned within *stringvar*.

Remarks

If *stringvar* is longer than *string*, **RSet** replaces any leftover characters in *stringvar* with spaces, back to its beginning.

Note **RSet** can't be used with user-defined types.

Visual Basic for Applications Reference

RSet Statement Example

This example uses the **RSet** statement to right align a string within a string variable.

```
Dim MyString
MyString = "0123456789"   ' Initialize string.
Rset MyString = "Right->"   ' MyString contains "   Right->".
```

Visual Basic Reference

Visual Studio 6.0

SavePicture Statement

See Also Example

Saves a graphic from the **Picture** or **Image** property of an object or control (if one is associated with it) to a file.

Syntax

SavePicture *picture*, *stringexpression*

The **SavePicture** statement syntax has these parts:

Part	Description
picture	**Picture** or **Image** control from which the graphics file is to be created.
stringexpression	Filename of the graphics file to save.

Remarks

If a graphic was loaded from a file to the **Picture** property of an object, either at design time or at run time, and its a bitmap, icon, metafile, or enhanced metafile, it's saved using the same format as the original file. If it is a GIF or JPEG file, it is saved as a bitmap file.

Graphics in an **Image** property are always saved as bitmap (.bmp) files regardless of their original format.

Visual Basic Reference

SavePicture Statement Example

This example uses the **SavePicture** statement to save a graphic drawn into a **Form** objects **Picture** property. To try this example, paste the code into the Declarations section of a **Form** object, and then run the example and click the **Form** object.

```
Private Sub Form_Click ()
   ' Declare variables.
   Dim CX, CY, Limit, Radius   as Integer, Msg as String
   ScaleMode = vbPixels   ' Set scale to pixels.
   AutoRedraw = True ' Turn on AutoRedraw.
   Width = Height   ' Change width to match height.
   CX = ScaleWidth / 2   ' Set X position.
   CY = ScaleHeight / 2   ' Set Y position.
   Limit = CX   ' Limit size of circles.
   For Radius = 0 To Limit   ' Set radius.
      Circle (CX, CY), Radius, RGB(Rnd * 255, Rnd * 255, Rnd * 255)
      DoEvents   ' Yield for other processing.
   Next Radius
   Msg = "Choose OK to save the graphics from this form "
   Msg = Msg & "to a bitmap file."
   MsgBox Msg
   SavePicture Image, "TEST.BMP"   ' Save picture to file.
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

SaveSetting Statement

See Also Example Specifics

Saves or creates an application entry in the application's entry in the Windows registry.

Syntax

SaveSetting *appname, section,* *key, setting*

The **SaveSetting** statement syntax has these named arguments:

Part	Description
appname	Required. String expression containing the name of the application or project to which the setting applies.
section	Required. String expression containing the name of the section where the key setting is being saved.
key	Required. String expression containing the name of the key setting being saved.
setting	Required. Expression containing the value that ***key*** is being set to.

Remarks

An error occurs if the key setting cant be saved for any reason.

Visual Basic for Applications Reference

SaveSetting Statement Example

The following example first uses the **SaveSetting** statement to make entries in the Windows registry (or .ini file on 16-bit Windows platforms) for the MyApp application, and then uses the **DeleteSetting** statement to remove them.

```
' Place some settings in the registry.
SaveSetting appname := "MyApp", section := "Startup", _
            key := "Top", setting := 75
SaveSetting "MyApp","Startup", "Left", 50
' Remove section and all its settings from registry.
DeleteSetting "MyApp", "Startup"
```

Visual Basic for Applications Reference

Visual Studio 6.0

Seek Statement

See Also Example Specifics

Sets the position for the next read/write operation within a file opened using the **Open** statement.

Syntax

Seek [**#**]*filenumber*, *position*

The **Seek** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
position	Required. Number in the range 1 2,147,483,647, inclusive, that indicates where the next read/write operation should occur.

Remarks

Record numbers specified in **Get** and **Put** statements override file positioning performed by **Seek**.

Performing a file-write operation after a **Seek** operation beyond the end of a file extends the file. If you attempt a **Seek** operation to a negative or zero position, an error occurs.

Visual Basic for Applications Reference

Seek Statement Example

This example uses the **Seek** statement to set the position for the next read or write within a file. This example assumes TESTFILE is a file containing records of the user-defined type Record.

```
Type Record   ' Define user-defined type.
   ID As Integer
   Name As String * 20
End Type
```

For files opened in Random mode, **Seek** sets the next record.

```
Dim MyRecord As Record, MaxSize, RecordNumber   ' Declare variables.
' Open file in random-file mode.
Open "TESTFILE" For Random As #1 Len = Len(MyRecord)
MaxSize = LOF(1) \ Len(MyRecord)   ' Get number of records in file.
' The loop reads all records starting from the last.
For RecordNumber = MaxSize To 1 Step - 1
   Seek #1, RecordNumber   ' Set position.
   Get #1, , MyRecord   ' Read record.
Next RecordNumber
Close #1   ' Close file.
```

For files opened in modes other than Random mode, **Seek** sets the byte position at which the next operation takes place. Assume TESTFILE is a file containing a few lines of text.

```
Dim MaxSize, NextChar, MyChar
Open "TESTFILE" For Input As #1   ' Open file for input.
MaxSize = LOF(1)   ' Get size of file in bytes.
' The loop reads all characters starting from the last.
For NextChar = MaxSize To 1 Step -1
   Seek #1, NextChar   ' Set position.
   MyChar = Input(1, #1)   ' Read character.
Next NextChar
Close #1   ' Close file.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Select Case Statement

See Also Example Specifics

Executes one of several groups of statements, depending on the value of an expression.

Syntax

Select Case *testexpression*
[**Case** *expressionlist-n*
[*statements-n*]] . . .
[**Case Else**
[*elsestatements*]]

End Select

The **Select Case** statement syntax has these parts:

Part	Description
testexpression	Required. Any numeric expression or string expression.
expressionlist-n	Required if a **Case** appears. Delimited list of one or more of the following forms: *expression*, *expression* **To** *expression*, **Is** *comparisonoperator expression*. The **To** keyword specifies a range of values. If you use the **To** keyword, the smaller value must appear before **To**. Use the **Is** keyword with comparison operators (except **Is** and **Like**) to specify a range of values. If not supplied, the **Is** keyword is automatically inserted.
statements-n	Optional. One or more statements executed if *testexpression* matches any part of *expressionlist-n*.
elsestatements	Optional. One or more statements executed if *testexpression* doesn't match any of the **Case** clause.

Remarks

If *testexpression* matches any **Case** *expressionlist* expression, the *statements* following that **Case** clause are executed up to the next **Case** clause, or, for the last clause, up to **End Select**. Control then passes to the statement following **End Select**. If *testexpression* matches an *expressionlist* expression in more than one **Case** clause, only the statements following the first match are executed.

The **Case Else** clause is used to indicate the *elsestatements* to be executed if no match is found between the *testexpression* and an *expressionlist* in any of the other **Case** selections. Although not required, it is a good idea to have a **Case Else** statement in your **Select Case** block to handle unforeseen *testexpression* values. If no **Case** *expressionlist* matches *testexpression* and there is no **Case Else** statement, execution continues at the statement following **End Select**.

You can use multiple expressions or ranges in each **Case** clause. For example, the following line is valid:

```
Case 1 To 4, 7 To 9, 11, 13, Is > MaxNumber
```

Note The **Is** comparison operator is not the same as the **Is** keyword used in the **Select Case** statement.

You also can specify ranges and multiple expressions for character strings. In the following example, **Case** matches strings that are exactly equal to `everything`, strings that fall between `nuts` and `soup` in alphabetic order, and the current value of `TestItem`:

```
Case "everything", "nuts" To "soup", TestItem
```

Select Case statements can be nested. Each nested **Select Case** statement must have a matching **End Select** statement.

Select Case Statement Example

This content is no longer actively maintained. It is provided as is, for anyone who may still be using these technologies, with no warranties or claims of accuracy with regard to the most recent product version or service release.

This example displays the name of the mail system installed on the computer.

```
Select Case Application.MailSystem
    Case Is = xlMAPI
        MsgBox "Mail system is Microsoft Mail"
    Case Is = xlPowerTalk
        MsgBox "Mail system is PowerTalk"
    Case Is = xlNoMailSystem
        MsgBox "No mail system installed"
End Select
```

This example displays a message box that indicates the location of the active cell in the PivotTable report.

```
Worksheets("Sheet1").Activate
Select Case ActiveCell.LocationInTable
Case Is = xlRowHeader
    MsgBox "Active cell is part of a row header"
Case Is = xlColumnHeader
    MsgBox "Active cell is part of a column header"
Case Is = xlPageHeader
    MsgBox "Active cell is part of a page header"
Case Is = xlDataHeader
    MsgBox "Active cell is part of a data header"
Case Is = xlRowItem
    MsgBox "Active cell is part of a row item"
Case Is = xlColumnItem
    MsgBox "Active cell is part of a column item"
Case Is = xlPageItem
    MsgBox "Active cell is part of a page item"
Case Is = xlDataItem
    MsgBox "Active cell is part of a data item"
Case Is = xlTableBody
    MsgBox "Active cell is part of the table body"
End Select
```

This example displays a message if the active cell on Sheet1 contains a cell error value. You can use this example as a framework for a cell-error-value error handler.

```
Worksheets("Sheet1").Activate
If IsError(ActiveCell.Value) Then
    errval = ActiveCell.Value
    Select Case errval
        Case CVErr(xlErrDiv0)
            MsgBox "#DIV/0! error"
        Case CVErr(xlErrNA)
            MsgBox "#N/A error"
        Case CVErr(xlErrName)
            MsgBox "#NAME? error"
        Case CVErr(xlErrNull)
            MsgBox "#NULL! error"
```

```
        Case CVErr(xlErrNum)
            MsgBox "#NUM! error"
        Case CVErr(xlErrRef)
            MsgBox "#REF! error"
        Case CVErr(xlErrValue)
            MsgBox "#VALUE! error"
        Case Else
            MsgBox "This should never happen!!"
    End Select
End If
```

Visual Basic for Applications Reference

Visual Studio 6.0

SendKeys Statement

See Also Example Specifics

Sends one or more keystrokes to the active window as if typed at the keyboard.

Syntax

SendKeys *string*[, *wait*]

The **SendKeys** statement syntax has these named arguments:

Part	Description
string	Required. String expression specifying the keystrokes to send.
Wait	Optional. Boolean value specifying the wait mode. If **False** (default), control is returned to the procedure immediately after the keys are sent. If **True**, keystrokes must be processed before control is returned to the procedure.

Remarks

Each key is represented by one or more characters. To specify a single keyboard character, use the character itself. For example, to represent the letter A, use `"A"` for ***string***. To represent more than one character, append each additional character to the one preceding it. To represent the letters A, B, and C, use `"ABC"` for ***string***.

The plus sign (**+**), caret (**^**), percent sign (**%**), tilde (**~**), and parentheses **( )** have special meanings to **SendKeys**. To specify one of these characters, enclose it within braces ({}). For example, to specify the plus sign, use {+}. Brackets ([ ]) have no special meaning to **SendKeys**, but you must enclose them in braces. In other applications, brackets do have a special meaning that may be significant when dynamic data exchange (DDE) occurs. To specify brace characters, use {{} and {}}.

To specify characters that aren't displayed when you press a key, such as ENTER or TAB, and keys that represent actions rather than characters, use the codes shown below:

Key	Code
BACKSPACE	{BACKSPACE}, {BS}, or {BKSP}
BREAK	{BREAK}
CAPS LOCK	{CAPSLOCK}

DEL or DELETE	{DELETE} or {DEL}
DOWN ARROW	{DOWN}
END	{END}
ENTER	{ENTER}or ~
ESC	{ESC}
HELP	{HELP}
HOME	{HOME}
INS or INSERT	{INSERT} or {INS}
LEFT ARROW	{LEFT}
NUM LOCK	{NUMLOCK}
PAGE DOWN	{PGDN}
PAGE UP	{PGUP}
PRINT SCREEN	{PRTSC}
RIGHT ARROW	{RIGHT}
SCROLL LOCK	{SCROLLLOCK}
TAB	{TAB}
UP ARROW	{UP}
F1	{F1}
F2	{F2}
F3	{F3}
F4	{F4}
F5	{F5}
F6	{F6}
F7	{F7}
F8	{F8}
F9	{F9}
F10	{F10}

F11	{F11}
F12	{F12}
F13	{F13}
F14	{F14}
F15	{F15}
F16	{F16}

To specify keys combined with any combination of the SHIFT, CTRL, and ALT keys, precede the key code with one or more of the following codes:

Key	Code
SHIFT	+
CTRL	^
ALT	%

To specify that any combination of SHIFT, CTRL, and ALT should be held down while several other keys are pressed, enclose the code for those keys in parentheses. For example, to specify to hold down SHIFT while E and C are pressed, use "+(EC)". To specify to hold down SHIFT while E is pressed, followed by C without SHIFT, use "+EC".

To specify repeating keys, use the form {key number}. You must put a space between key and number. For example, {LEFT 42} means press the LEFT ARROW key 42 times; {h 10} means press H 10 times.

Note You can't use **SendKeys** to send keystrokes to an application that is not designed to run in Microsoft Windows. **Sendkeys** also can't send the PRINT SCREEN key {PRTSC} to any application.

Visual Basic for Applications Reference

SendKeys Statement Example

This example uses the **Shell** function to run the Calculator application included with Microsoft Windows. It uses the **SendKeys** statement to send keystrokes to add some numbers, and then quit the Calculator. (To see the example, paste it into a procedure, then run the procedure. Because **AppActivate** changes the focus to the Calculator application, you can't single step through the code.).

```
Dim ReturnValue, I
ReturnValue = Shell("CALC.EXE", 1)   ' Run Calculator.
AppActivate ReturnValue    ' Activate the Calculator.
For I = 1 To 100   ' Set up counting loop.
   SendKeys I & "{+}", True   ' Send keystrokes to Calculator
Next I   ' to add each value of I.
SendKeys "=", True   ' Get grand total.
SendKeys "%{F4}", True   ' Send ALT+F4 to close Calculator.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Set Statement

See Also Example Specifics

Assigns an object reference to a variable or property.

Syntax

Set *objectvar* = {[**New**] *objectexpression* | **Nothing**}

The **Set** statement syntax has these parts:

Part	Description
objectvar	Required. Name of the variable or property; follows standard variable naming conventions.
New	Optional. **New** is usually used during declaration to enable implicit object creation. When **New** is used with **Set**, it creates a new instance of the class. If *objectvar* contained a reference to an object, that reference is released when the new one is assigned. The **New** keyword can't be used to create new instances of any intrinsic data type and can't be used to create dependent objects.
objectexpression	Required. Expression consisting of the name of an object, another declared variable of the same object type, or a function or method that returns an object of the same object type.
Nothing	Optional. Discontinues association of *objectvar* with any specific object. Assigning **Nothing** to *objectvar* releases all the system and memory resources associated with the previously referenced object when no other variable refers to it.

Remarks

To be valid, *objectvar* must be an object type consistent with the object being assigned to it.

The **Dim**, **Private**, **Public**, **ReDim**, and **Static** statements only declare a variable that refers to an object. No actual object is referred to until you use the **Set** statement to assign a specific object.

The following example illustrates how **Dim** is used to declare an array with the type `Form1`. No instance of `Form1` actually exists. **Set** then assigns references to new instances of `Form1` to the `myChildForms` variable. Such code might be used to create child forms in an MDI application.

```
Dim myChildForms(1 to 4) As Form1
Set myChildForms(1) = New Form1
Set myChildForms(2) = New Form1
```

```
Set myChildForms(3) = New Form1
Set myChildForms(4) = New Form1
```

Generally, when you use **Set** to assign an object reference to a variable, no copy of the object is created for that variable. Instead, a reference to the object is created. More than one object variable can refer to the same object. Because such variables are references to the object rather than copies of the object, any change in the object is reflected in all variables that refer to it. However, when you use the **New** keyword in the **Set** statement, you are actually creating an instance of the object.

Set Statement Example

This content is no longer actively maintained. It is provided as is, for anyone who may still be using these technologies, with no warranties or claims of accuracy with regard to the most recent product version or service release.

This example adds a new worksheet to the active workbook and then sets the name of the worksheet.

```
Set newSheet = Worksheets.Add
newSheet.Name = "1995 Budget"
```

This example creates a new worksheet and then inserts into it a list of all the names in the active workbook, including their formulas in A1-style notation in the language of the user.

```
Set newSheet = ActiveWorkbook.Worksheets.Add
i = 1
For Each nm In ActiveWorkbook.Names
    newSheet.Cells(i, 1).Value = nm.NameLocal
    newSheet.Cells(i, 2).Value = "'" & nm.RefersToLocal
    i = i + 1
Next
```

This documentation is archived and is not being maintained.

Visual Basic for Applications Reference

Visual Studio 6.0

SetAttr Statement

See Also Example Specifics

Sets attribute information for a file.

Syntax

SetAttr *pathname, attributes*

The **SetAttr** statement syntax has these named arguments:

Part	Description
pathname	Required. String expression that specifies a file name may include directory or folder, and drive.
attributes	Required. Constant or numeric expression, whose sum specifies file attributes.

Settings

The ***attributes*** argument settings are:

Constant	Value	Description
vbNormal	0	Normal (default).
vbReadOnly	1	Read-only.
vbHidden	2	Hidden.
vbSystem	4	System file.
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

A run-time error occurs if you try to set the attributes of an open file.

Visual Basic for Applications Reference

SetAttr Statement Example

This example uses the **SetAttr** statement to set attributes for a file.

```
SetAttr "TESTFILE", vbHidden   ' Set hidden attribute.
SetAttr "TESTFILE", vbHidden + vbReadOnly   ' Set hidden and read-only
   ' attributes.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Static Statement

See Also Example Specifics

Used at procedure level to declare variables and allocate storage space. Variables declared with the **Static** statement retain their values as long as the code is running.

Syntax

Static *varname*[**(**[*subscripts*]**)**] [**As** [**New**] *type*] [**,** *varname*[**(**[*subscripts*]**)**] [**As** [**New**] *type*]] . . .

The **Static** statement syntax has these parts:

Part	Description
varname	Required. Name of the variable; follows standard variable naming conventions.
subscripts	Optional. Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax: [*lower* **To**] *upper* [**,**[*lower* **To**] *upper*] . . . When not explicitly stated in *lower*, the lower bound of an array is controlled by the **Option Base** statement. The lower bound is zero if no **Option Base** statement is present.
New	Optional. Keyword that enables implicit creation of an object. If you use **New** when declaring the object variable, a new instance of the object is created on first reference to it, so you don't have to use the **Set** statement to assign the object reference. The **New** keyword can't be used to declare variables of any intrinsic data type and can't be used to declare instances of dependent objects.
type	Optional. Data type of the variable; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String, (for variable-length strings), **String** * *length* (for fixed-length strings), Object, Variant, a user-defined type, or an object type. Use a separate **As** *type* clause for each variable being defined.

Remarks

Once module code is running, variables declared with the **Static** statement retain their value until the module is reset or restarted. In class modules, variables declared with the **Static** statement retain their value in each class instance until that instance is destroyed. In form modules, static variables retain their value until the form is closed. Use the **Static** statement in nonstatic procedures to explicitly declare variables that are visible only within the procedure, but whose lifetime is the same as the module in which the procedure is defined.

Use a **Static** statement within a procedure to declare the data type of a variable that retains its value between procedure calls. For example, the following statement declares a fixed-size array of integers:

```
Static EmployeeNumber(200) As Integer
```

The following statement declares a variable for a new instance of a worksheet:

```
Static X As New Worksheet
```

If the **New** keyword isn't used when declaring an object variable, the variable that refers to the object must be assigned an existing object using the **Set** statement before it can be used. Until it is assigned an object, the declared object variable has the special value **Nothing**, which indicates that it doesn't refer to any particular instance of an object. When you use the **New** keyword in the declaration, an instance of the object is created on the first reference to the object.

If you don't specify a data type or object type, and there is no **Def***type* statement in the module, the variable is **Variant** by default.

Note The **Static** statement and the **Static** keyword are similar, but used for different effects. If you declare a procedure using the **Static** keyword (as in `Static Sub CountSales ()`), the storage space for all local variables within the procedure is allocated once, and the value of the variables is preserved for the entire time the program is running. For nonstatic procedures, storage space for variables is allocated each time the procedure is called and released when the procedure is exited. The **Static** statement is used to declare specific variables within nonstatic procedures to preserve their value for as long as the program is running.

When variables are initialized, a numeric variable is initialized to 0, a variable-length string is initialized to a zero-length string (""), and a fixed-length string is filled with zeros. **Variant** variables are initialized to Empty. Each element of a user-defined type variable is initialized as if it were a separate variable.

Note When you use **Static** statements within a procedure, put them at the beginning of the procedure with other declarative statements such as **Dim**.

Static Statement Example

This content is no longer actively maintained. It is provided as is, for anyone who may still be using these technologies, with no warranties or claims of accuracy with regard to the most recent product version or service release.

This example uses the worksheet function **Pmt** to calculate a home mortgage loan payment. Note that this example uses the **InputBox** method instead of the **InputBox** function so that the method can perform type checking. The **Static** statements cause Visual Basic to retain the values of the three variables; these are displayed as default values the next time you run the example.

```
Static loanAmt
Static loanInt
Static loanTerm
loanAmt = Application.InputBox _
    (Prompt:="Loan amount (100,000 for example)", _
        Default:=loanAmt, Type:=1)
loanInt = Application.InputBox _
    (Prompt:="Annual interest rate (8.75 for example)", _
        Default:=loanInt, Type:=1)
loanTerm = Application.InputBox _
    (Prompt:="Term in years (30 for example)", _
        Default:=loanTerm, Type:=1)
payment = Application.Pmt(loanInt / 1200, loanTerm * 12, loanAmt)
MsgBox "Monthly payment is " & Format(payment, "Currency")
```

Visual Basic for Applications Reference

Visual Studio 6.0

Stop Statement

See Also Example Specifics

Suspends execution.

Syntax

Stop

Remarks

You can place **Stop** statements anywhere in procedures to suspend execution. Using the **Stop** statement is similar to setting a breakpoint in the code.

The **Stop** statement suspends execution, but unlike **End**, it doesn't close any files or clear variables, unless it is in a compiled executable (.exe) file.

Visual Basic for Applications Reference

Stop Statement Example

This example uses the **Stop** statement to suspend execution for each iteration through the **For...Next** loop.

```
Dim I
For I = 1 To 10   ' Start For...Next loop.
   Debug.Print I   ' Print I to the Immediate window.
   Stop   ' Stop during each iteration.
Next I
```

Visual Basic for Applications Reference

Visual Studio 6.0

Sub Statement

See Also Example Specifics

Declares the name, arguments, and code that form the body of a **Sub** procedure.

Syntax

[**Private** | **Public** | **Friend**] [**Static**] **Sub** *name* [**(***arglist***)**]
[*statements*]
[**Exit Sub**]
[*statements*]

End Sub

The **Sub** statement syntax has these parts:

Part	Description
Public	Optional. Indicates that the **Sub** procedure is accessible to all other procedures in all modules. If used in a module that contains an **Option Private** statement, the procedure is not available outside the project.
Private	Optional. Indicates that the **Sub** procedure is accessible only to other procedures in the module where it is declared.
Friend	Optional. Used only in a class module. Indicates that the **Sub** procedure is visible throughout the project, but not visible to a controller of an instance of an object.
Static	Optional. Indicates that the **Sub** procedure's local variables are preserved between calls. The **Static** attribute doesn't affect variables that are declared outside the **Sub**, even if they are used in the procedure.
name	Required. Name of the **Sub**; follows standard variable naming conventions.
arglist	Optional. List of variables representing arguments that are passed to the **Sub** procedure when it is called. Multiple variables are separated by commas.
statements	Optional. Any group of statements to be executed within the **Sub** procedure.

The *arglist* argument has the following syntax and parts:

[**Optional**] [**ByVal** | **ByRef**] [**ParamArray**] *varname*[**( )**] [**As** *type*] [**=** *defaultvalue*]

Part	Description
Optional	Optional. Keyword indicating that an argument is not required. If used, all subsequent arguments in *arglist* must also be optional and declared using the **Optional** keyword. **Optional** can't be used for any argument if **ParamArray** is used.
ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Optional. Indicates that the argument is passed by reference. **ByRef** is the default in Visual Basic.
ParamArray	Optional. Used only as the last argument in *arglist* to indicate that the final argument is an **Optional** array of **Variant** elements. The **ParamArray** keyword allows you to provide an arbitrary number of arguments. **ParamArray** can't be used with **ByVal**, **ByRef**, or **Optional**.
varname	Required. Name of the variable representing the argument; follows standard variable naming conventions.
type	Optional. Data type of the argument passed to the procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (variable-length only), Object, Variant, or a specific object type. If the parameter is not **Optional**, a user-defined type may also be specified.
defaultvalue	Optional. Any constant or constant expression. Valid for **Optional** parameters only. If the type is an **Object**, an explicit default value can only be **Nothing**.

Remarks

If not explicitly specified using **Public**, **Private**, or **Friend**, **Sub** procedures are public by default. If **Static** isn't used, the value of local variables is not preserved between calls. The **Friend** keyword can only be used in class modules. However, **Friend** procedures can be accessed by procedures in any module of a project. A **Friend** procedure doesn't appear in the type library of its parent class, nor can a **Friend** procedure be late bound.

Caution **Sub** procedures can be recursive; that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow. The **Static** keyword usually is not used with recursive **Sub** procedures.

All executable code must be in procedures. You can't define a **Sub** procedure inside another **Sub**, **Function**, or **Property** procedure.

The **Exit Sub** keywords cause an immediate exit from a **Sub** procedure. Program execution continues with the statement following the statement that called the **Sub** procedure. Any number of **Exit Sub** statements can appear anywhere in a **Sub** procedure.

Like a **Function** procedure, a **Sub** procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a **Function** procedure, which returns a value, a **Sub** procedure can't be used in an expression.

You call a **Sub** procedure using the procedure name followed by the argument list. See the **Call** statement for specific information on how to call **Sub** procedures.

Variables used in **Sub** procedures fall into two categories: those that are explicitly declared within the procedure and those that are not. Variables that are explicitly declared in a procedure (using **Dim** or the equivalent) are always local to the procedure. Variables that are used but not explicitly declared in a procedure are also local unless they are explicitly declared at some higher level outside the procedure.

Caution A procedure can use a variable that is not explicitly declared in the procedure, but a naming conflict can occur if anything you defined at the module level has the same name. If your procedure refers to an undeclared variable that has the same name as another procedure, constant or variable, it is assumed that your procedure is referring to that module-level

name. To avoid this kind of conflict, explicitly declare variables. You can use an **Option Explicit** statement to force explicit declaration of variables.

Note You can't use **GoSub**, **GoTo**, or **Return** to enter or exit a **Sub** procedure.

Visual Basic for Applications Reference

Sub Statement Example

This example uses the **Sub** statement to define the name, arguments, and code that form the body of a **Sub** procedure.

```
' Sub procedure definition.
' Sub procedure with two arguments.
Sub SubComputeArea(Length, TheWidth)
   Dim Area As Double   ' Declare local variable.
   If Length = 0 Or TheWidth = 0 Then
   ' If either argument = 0.
      Exit Sub   ' Exit Sub immediately.
   End If
   Area = Length * TheWidth   ' Calculate area of rectangle.
   Debug.Print Area   ' Print Area to Debug window.
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

Time Statement

See Also Example Specifics

Sets the system time.

Syntax

Time = *time*

The required *time* argument is any numeric expression, string expression, or any combination, that can represent a time.

Remarks

If *time* is a string, **Time** attempts to convert it to a time using the time separators you specified for your system. If it can't be converted to a valid time, an error occurs.

Visual Basic for Applications Reference

Time Statement Example

This example uses the **Time** statement to set the computer system time to a user-defined time.

```
Dim MyTime
MyTime = #4:35:17 PM#   ' Assign a time.
Time = MyTime   ' Set system time to MyTime.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Type Statement

See Also Example Specifics

Used at module level to define a user-defined data type containing one or more elements.

Syntax

[**Private** | **Public**] **Type** *varname*
elementname [**(**[*subscripts*]**)**] **As** *type*
[*elementname* [**(**[*subscripts*]**)**] **As** *type*]
. . .

End Type

The **Type** statement syntax has these parts:

Part	Description
Public	Optional. Used to declare user-defined types that are available to all procedures in all modules in all projects.
Private	Optional. Used to declare user-defined types that are available only within the module where the declaration is made.
varname	Required. Name of the user-defined type; follows standard variable naming conventions.
elementname	Required. Name of an element of the user-defined type. Element names also follow standard variable naming conventions, except that keywords can be used.
subscripts	When not explicitly stated in *lower*, the lower bound of an array is controlled by the **Option Base** statement. The lower bound is zero if no **Option Base** statement is present.
type	Required. Data type of the element; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (for variable-length strings), **String** * *length* (for fixed-length strings), Object, Variant, another user-defined type, or an object type.

Remarks

The **Type** statement can be used only at module level. Once you have declared a user-defined type using the **Type** statement, you can declare a variable of that type anywhere within the scope of the declaration. Use **Dim**, **Private**, **Public**, **ReDim**, or **Static** to declare a variable of a user-defined type.

In standard modules and class modules, user-defined types are public by default. This visibility can be changed using the **Private** keyword.

Line numbers and line labels aren't allowed in **Type...End Type** blocks.

User-defined types are often used with data records, which frequently consist of a number of related elements of different data types.

The following example shows the use of fixed-size arrays in a user-defined type:

```
Type StateData
   CityCode (1 To 100) As Integer    ' Declare a static array.
   County As String * 30
End Type

Dim Washington(1 To 100) As StateData
```

In the preceding example, `StateData` includes the `CityCode` static array, and the record `Washington` has the same structure as `StateData`.

When you declare a fixed-size array within a user-defined type, its dimensions must be declared with numeric literals or constants rather than variables.

Visual Basic for Applications Reference

Type Statement Example

This example uses the **Type** statement to define a user-defined data type. The **Type** statement is used at the module level only. If it appears in a class module, a **Type** statement must be preceded by the keyword **Private**.

```
Type EmployeeRecord   ' Create user-defined type.
   ID As Integer   ' Define elements of data type.
   Name As String * 20
   Address As String * 30
   Phone As Long
   HireDate As Date
End Type
Sub CreateRecord()
   Dim MyRecord As EmployeeRecord   ' Declare variable.

   ' Assignment to EmployeeRecord variable must occur in a procedure.
   MyRecord.ID = 12003   ' Assign a value to an element.
End Sub
```

Visual Basic Reference

Visual Studio 6.0

Unload Statement

See Also Example

Unloads a form or control from memory.

Syntax

Unload *object*

The *object* placeholder is the name of a **Form** object or control array element to unload.

Remarks

Unloading a form or control may be necessary or expedient in some cases where the memory used is needed for something else, or when you need to reset properties to their original values.

Before a form is unloaded, the Query_Unload event procedure occurs, followed by the Form_Unload event procedure. Setting the *cancel* argument to **True** in either of these events prevents the form from being unloaded. For **MDIForm** objects, the **MDIForm** object's Query_Unload event procedure occurs, followed by the Query_Unload event procedure and Form_Unload event procedure for each MDI child form, and finally the **MDIForm** object's Form_Unload event procedure.

When a form is unloaded, all controls placed on the form at run time are no longer accessible. Controls placed on the form at design time remain intact; however, any run-time changes to those controls and their properties are lost when the form is reloaded. All changes to form properties are also lost. Accessing any controls on the form causes it to be reloaded.

Note When a form is unloaded, only the displayed component is unloaded. The code associated with the form module remains in memory.

Only control array elements added to a form at run time can be unloaded with the **Unload** statement. The properties of unloaded controls are reinitialized when the controls are reloaded.

Visual Basic Reference

Unload Statement Example

This example uses the **Unload** statement to unload a **Form** object. To try this example, paste the code into the Declarations section of a **Form** object, and then run the example and click the **Form** object.

```
Private Sub Form_Click ()
   Dim Answer, Msg   ' Declare variable.
   Unload Form1   ' Unload form.
   Msg = "Form1 has been unloaded. Choose Yes to load and "
   Msg = Msg & "display the form. Choose No to load the form "
   Msg = Msg & "and leave it invisible."
   Answer = MsgBox(Msg, vbYesNo)   ' Get user response.
   If Answer = vbYes Then   ' Evaluate answer.
      Show   ' If Yes, show form.
   Else
      Load Form1   ' If No, just load it.
      Msg = "Form1 is now loaded. Choose OK to display it."
      MsgBox Msg   ' Display message.
      Show   ' Show form.
   End If
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

While...Wend Statement

See Also Example Specifics

Executes a series of statements as long as a given condition is **True**.

Syntax

While *condition*
[*statements*]

Wend

The **While...Wend** statement syntax has these parts:

Part	Description
condition	Required. Numeric expression or string expression that evaluates to **True** or **False**. If *condition* is Null, *condition* is treated as **False**.
statements	Optional. One or more statements executed while condition is **True**.

Remarks

If *condition* is **True**, all *statements* are executed until the **Wend** statement is encountered. Control then returns to the **While** statement and *condition* is again checked. If *condition* is still **True**, the process is repeated. If it is not **True**, execution resumes with the statement following the **Wend** statement.

While...Wend loops can be nested to any level. Each **Wend** matches the most recent **While**.

Tip The **Do...Loop** statement provides a more structured and flexible way to perform looping.

Visual Basic for Applications Reference

While...Wend Statement Example

This example uses the **While...Wend** statement to increment a counter variable. The statements in the loop are executed as long as the condition evaluates to **True**.

```
Dim Counter
Counter = 0   ' Initialize variable.
While Counter < 20   ' Test value of Counter.
   Counter = Counter + 1   ' Increment Counter.
Wend   ' End While loop when Counter > 19.
Debug.Print Counter   ' Prints 20 in the Immediate window.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Width # Statement

See Also Example Specifics

Assigns an output line width to a file opened using the **Open** statement.

Syntax

Width #*filenumber*, *width*

The **Width #** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
width	Required. Numeric expression in the range 0255, inclusive, that indicates how many characters appear on a line before a new line is started. If *width* equals 0, there is no limit to the length of a line. The default value for *width* is 0.

Visual Basic for Applications Reference

Width # Statement Example

This example uses the **Width #** statement to set the output line width for a file.

```
Dim I
Open "TESTFILE" For Output As #1   ' Open file for output.
VBA.Width 1, 5   ' Set output line width to 5.
For I = 0 To 9   ' Loop 10 times.
   Print #1, Chr(48 + I);   ' Prints five characters per line.
Next I
Close #1   ' Close file.
```

Visual Basic for Applications Reference

Visual Studio 6.0

With Statement

See Also Example Specifics

Executes a series of statements on a single object or a user-defined type.

Syntax

With *object*
[*statements*]

End With

The **With** statement syntax has these parts:

Part	Description
object	Required. Name of an object or a user-defined type.
statements	Optional. One or more statements to be executed on *object*.

Remarks

The **With** statement allows you to perform a series of statements on a specified object without requalifying the name of the object. For example, to change a number of different properties on a single object, place the property assignment statements within the **With** control structure, referring to the object once instead of referring to it with each property assignment. The following example illustrates use of the **With** statement to assign values to several properties of the same object.

```
With MyLabel
   .Height = 2000
   .Width = 2000
   .Caption = "This is MyLabel"
End With
```

Note Once a **With** block is entered, *object* can't be changed. As a result, you can't use a single **With** statement to affect a number of different objects.

You can nest **With** statements by placing one **With** block within another. However, because members of outer **With** blocks are masked within the inner **With** blocks, you must provide a fully qualified object reference in an inner **With** block to any member of an object in an outer **With** block.

Note In general, it's recommended that you don't jump into or out of **With** blocks. If statements in a **With** block are executed, but either the **With** or **End With** statement is not executed, a temporary variable containing a reference to the object remains in memory until you exit the procedure.

With Statement Example

This example creates a formatted multiplication table in cells A1:K11 on Sheet1.

```
Set dataTableRange = Worksheets("Sheet1").Range("A1:K11")
Set rowInputCell = Worksheets("Sheet1").Range("A12")
Set columnInputCell = Worksheets("Sheet1").Range("A13")

Worksheets("Sheet1").Range("A1").Formula = "=A12*A13"
For i = 2 To 11
    Worksheets("Sheet1").Cells(i, 1) = i - 1
    Worksheets("Sheet1").Cells(1, i) = i - 1
Next i
dataTableRange.Table rowInputCell, columnInputCell
With Worksheets("Sheet1").Range("A1").CurrentRegion
    .Rows(1).Font.Bold = True
    .Columns(1).Font.Bold = True
    .Columns.AutoFit
End With
```

Visual Basic for Applications Reference

Visual Studio 6.0

Write # Statement

See Also Example Specifics

Writes data to a sequential file.

Syntax

Write #*filenumber*, [*outputlist*]

The **Write #** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
outputlist	Optional. One or more comma-delimited numeric expressions or string expressions to write to a file.

Remarks

Data written with **Write #** is usually read from a file with **Input #**.

If you omit *outputlist* and include a comma after *filenumber*, a blank line is printed to the file. Multiple expressions can be separated with a space, a semicolon, or a comma. A space has the same effect as a semicolon.

When **Write #** is used to write data to a file, several universal assumptions are followed so the data can always be read and correctly interpreted using **Input #**, regardless of locale:

- Numeric data is always written using the period as the decimal separator.
- For Boolean data, either `#TRUE#` or `#FALSE#` is printed. The **True** and **False** keywords are not translated, regardless of locale.
- Date data is written to the file using the universal date format. When either the date or the time component is missing or zero, only the part provided gets written to the file.
- Nothing is written to the file if *outputlist* data is Empty. However, for Null data, `#NULL#` is written.
- If *outputlist* data is **Null** data, `#NULL#` is written to the file.
- For **Error** data, the output appears as `#ERROR errorcode#`. The **Error** keyword is not translated, regardless of locale.

Unlike the **Print #** statement, the **Write #** statement inserts commas between items and quotation marks around strings as they are written to the file. You don't have to put explicit delimiters in the list. **Write #** inserts a newline character, that is, a

carriage returnlinefeed (**Chr(**13**)** + **Chr(**10**)**), after it has written the final character in *outputlist* to the file.

Note You should not write strings that contain embedded quotation marks, for example, `"1,2""X"` for use with the **Input #** statement: **Input #** parses this string as two complete and separate strings.

Visual Basic for Applications Reference

Write # Statement Example

This example uses the **Write #** statement to write raw data to a sequential file.

```
Open "TESTFILE" For Output As #1   ' Open file for output.
Write #1, "Hello World", 234   ' Write comma-delimited data.
Write #1,   ' Write blank line.

Dim MyBool, MyDate, MyNull, MyError
' Assign Boolean, Date, Null, and Error values.
MyBool = False : MyDate = #February 12, 1969# : MyNull = Null
MyError = CVErr(32767)
' Boolean data is written as #TRUE# or #FALSE#. Date literals are
' written in universal date format, for example, #1994-07-13#
 'represents July 13, 1994. Null data is written as #NULL#.
' Error data is written as #ERROR errorcode#.
Write #1, MyBool ; " is a Boolean value"
Write #1, MyDate ; " is a date"
Write #1, MyNull ; " is a null value"
Write #1, MyError ; " is an error value"
Close #1   ' Close file.
```