ESKER SmarTerm®

Macro Guide

Esker® SOFTWARE

Esker SOFTWARE

Extending the Reach of Information

Esker S.A., 10 rue des Émeraudes, 69006 Lyon, France
Tel: +33 (0)4.72.83.46.46 ♦ Fax: +33 (0)4.72.83.46.40 ♦ info@esker.fr ♦ www.esker.fr

Esker, Inc., 1212 Deming Way, Suite 350, Madison, WI 53717-1984 USA
Tel: +1.608.828.6000 ♦ Fax: +1.608.828.6001 ♦ info@esker.com ♦ www.esker.com

Esker Australia Pty Ltd. (Sydney) ♦ Tel: +61 2 8596 5100 ♦ info@esker.com.au ♦ www.esker.com.au
Esker GmbH (München) ♦ Tel: +49 (0) 89 700 887 0 ♦ info@esker.de ♦ www.esker.de
Esker Italia SRL (Milano) ♦ Tel: +39 02 89 20 03 03 ♦ info@esker.it ♦ www.esker.it
Esker Ibérica, S.L. (Madrid) ♦ Tel: +34 91 552 9265 ♦ info@esker.es ♦ www.esker.es
Esker UK Ltd. (Derby) ♦ Tel: +44 1332 54 8181 ♦ info@esker.co.uk ♦ www.esker.co.uk

Contents

Introduction

The SmarTerm macro language is a powerful Visual-Basic compatible macro language tailored especially for use with SmarTerm. This ***Macro Guide*** provides a brief overview of and tutorial for the language, plus comprehensive descriptions of all the features of the language. The initial chapters cover basic features of the languages, such as data types, operators, expressions, compilation control features, and keywords. Subsequent chapters are an a-to-z reference of all macro language statements and functions, as well as all object properties and methods. This long section is followed by two short appendices, one listing equivalents to the older Persoft Script Language (PSL), and the other listing the numeric error messages you might receive from the macro compiler.

Note All information covered in this manual is also available in the online help system. In addition, the online version of this manual may be more up to date than the printed version.

Throughout this manual we use the following conventions:

- Examples are shown in a **`type-in font`**.
- Optional parameters are enclosed in square brackets: **`[ ]`**.
- Named parameters are *`italicized`*.
- Options in a series are separated with the pipe character: **`|`**.
- If you can specify multiple similar parameters, only the first and last are specified, and the intermediate parameters are indicated with an ellipsis: `...`.

Macro Features Listed by Purpose

File Transfer

Character and String Manipulation

Drive, Folder, and File Access

Keywords, Data Types, Operators, and Expressions

Host Connections

Numeric, Math, and Accounting Functions

Macro Control and Compilation

Application and Session Features

Operating System Control

User Interaction

Time and Date Access

Objects

SQL Access

DDE Access

CreateObject (function) 188
DDEExecute (statement) 205
DDEInitiate (function) 205
DDEPoke (statement) 206
DDERequest, DDERequest$ (functions) 206
DDESend 207
DDETerminate 208
DDETerminateAll 208
DDETimeout 209
GetObject (function) 290
CreateObject (function) 188
DDEExecute (statement) 205
DDEInitiate (function) 205

Recording and Running Macros

When you start up SmarTerm, select Tools>Macros and click Record, you start a macro recorder that:

- Records what you do in a file
- Automatically writes it in the SmarTerm macro language
- Documents what it records

You then can replay the macro or edit it using the macro editor.

When you record a macro, you might keep in mind that the Toolbox doesn't record every action you perform. Instead, it analyzes your actions and records those that can be performed with macro commands. The recorder also looks for incoming prompts and stores outgoing keystrokes.

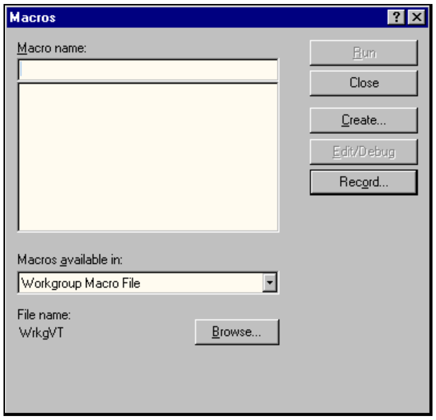
For example, SmarTerm provides a full range of file transfer capabilities. Therefore, when you record a file transfer, the entire process is recorded. However, the macro language does not support editing a macro in the macro editor, so you cannot record that sort of task in a macro.

This chapter describes how to record and use macros. More macro information follows in the next two chapters, "Creating Macros" on page 25 and "Programming Macros" on page 41.

Recording macros

To record a macro:

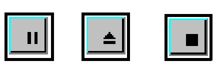
1. Select Tools>Macros. The Macros dialog appears:



Select the file where the macro is to be stored.

2. Type a name for your macro. Don't include spaces in the name. To replace an existing macro, select the name from the list.
3. Click Record. The Start Recording dialog appears, allowing you to review the macro name you just typed. If you use an existing macro name, SmarTerm asks whether you want to overwrite that macro. Agree, or change the name, and then click OK. Your session reappears with the word "Record" in the status bar and a set of buttons that allow you to control the recording process.
4. Perform the steps you want to record.

   At any time you can click the Pause button to pause the recording or the Abort button to abort the recording.



Pause Abort Stop

5. When you are finished recording the macro, click the Stop button to save the macro. If you entered passwords while recording the macro, a Password Handling dialog appears. You can choose to store the password in the macro or to require the macro to prompt for the password each time you run it.

Running macros

To test a macro, select Tools>Macros, select the file and macro you want to run, and click Run. You can also assign a macro to a keystroke, a SmartMouse action, or a SmarTerm button. Follow these instructions in the online Help for the tool which you want to use.

What can go wrong?

The Toolbox can't record everything you do in a macro. For example, you might record a macro that includes a specific response from the host. If you run the macro again and get a different response from the host, the macro may get out of sync. If this happens, stop the macro and then try running it again to see if the same thing happens. If the host consistently produces the same new response, you can record the macro again to put the new host response into the macro. If the problem is that you cannot predict the host's response, you may have to edit macro to allow for multiple responses from the host. See the chapter on Creating Macros for information on editing macros.

Running PSL Scripts

Before SmarTerm 6.0, the SmarTerm products relied on the Persoft Script Language (PSL). Since then, the Visual Basic compatible SmarTerm Macro Language has replaced PSL. If you are upgrading old sessions to the current version, SmarTerm automatically converts most of the old PSL scripts, those associated with:

- Automatic login and logout
- SmartMouse actions
- Keyboard mappings

Note Only old button palettes and toolbars require you to run a converter. In the online help, under Tools>Toolbar or Tools>SmarTerm Buttons, you’ll find a Toolbar and Button Palette Converter book with conversion instructions.

Creating Macros

The SmarTerm macro language is an implementation of VisualBasic for Applications (VBA) especially tailored for use with SmarTerm. The previous chapter described how to use the macro recorder to record and play back simple macros (see “Recording and Running Macros” on page 21). There are times, however, when the tasks that you want to accomplish are too complicated for simple recording, so SmarTerm comes with an integrated editor and debugger that allow you to write more complex macros. This and the following chapter explain how to do this.

This chapter briefly describes the features of the SmarTerm macro language and explains how macros are organized in SmarTerm. The next chapter describes how to program macros for a variety of basic tasks (see “Programming Macros” on page 41), and the last chapter explains how to best use macros when you need the sophistication and flexibility required in a large organization.

Before getting started, please note that these chapters, although constituting a sort of macro tutorial, are probably not appropriate if you have never programmed before, or if you are not familiar with SmarTerm. This tutorial does not assume complete mastery of either of these topics, but it does require at least some familiarity with topics such as looping constructs, arrays, functions, data typing, and so forth, as well as a sense of what one does with terminal emulation software.

Features and organization

The SmarTerm Macro Language provides you with customizable control over most aspects of host communication. Commands in the language let you:

- Make host connections using all of the communication methods supported by SmarTerm
- Modify the settings of all of the emulation types supported by SmarTerm
- Transfer files using all of the file transfer methods supported by SmarTerm
- Build Windows-style user interfaces for your macros using the integrated visual dialog editor
- Have access to the most important operating system functions such as disk and file access, OLE (Object Linking and Embedding) automation, and so forth

You may be familiar with another macro language that organizes macros in a particular way. For example, many macro languages simply store each macro in a file, and allow you to open and run one or another macro file. SmarTerm, like other Windows applications that support a VBA-based macro language (such as Microsoft Word), uses a somewhat more complicated system. In part this is in recognition of the greater flexibility required by emulation software (since we can't know what host applications you may use with SmarTerm). However, it is also in response to the needs of large, server-oriented sites that need more sophisticated tools to support the needs of their users. Later in this chapter we describe how macros are organized, and provide some tips to help you take advantage of this organization.

Macro syntax

A single macro is simply a block of text with macro commands in it stored in some location accessible to SmarTerm (called a macro *module*). Macros may be *subroutines* (which carry out commands but do not return a result that can be assigned to a variable) or *functions* (subroutines which do return a result that can be assigned to a variable). In this chapter, unless specifically stated otherwise, you may assume that any reference to "subroutine" can be expanded to include functions as well.

The text for a macro must have:

- A first line that is `Sub` for a subroutine or `Function` for a function, followed by the name of the subroutine or function. This name must follow the conventions described in the online help for subroutines and functions.
- **For subroutines only:** If you want the macro to be selectable from the Tools>Macros dialog when the module is loaded, the second line must begin `'!` (a single quotation mark followed by an exclamation point). If you want a description of the macro to appear in the Macros dialog, put the text you want after the `'!`. You can have up to three lines of 66 characters each for the description, each beginning with `'!`. SmarTerm puts as much text as possible on each of the three lines, even if you insert carriage returns.

Note Functions do not appear in the Tools>Macros dialog, even if they have the `'!` description line.

- One or more lines of text containing control statements to carry out the macro's purpose. Each line is considered to end when the compiler encounters a comment or the carriage-return linefeed combination that ends a line in an ASCII text file. If you need to, you can continue a line of code onto the next line of the macro by preceding the carriage-return with an underscore (_), the line continuation character. Any line or section of a line that has been commented (see "Adding comments to macros" on page 45) is ignored by the compiler.
- A last line marking the end of the macro that corresponds to the first, either `End Sub` or `End Function`.

For example, a macro containing file transfer commands to fetch a weekly status report might look something like this in the module:

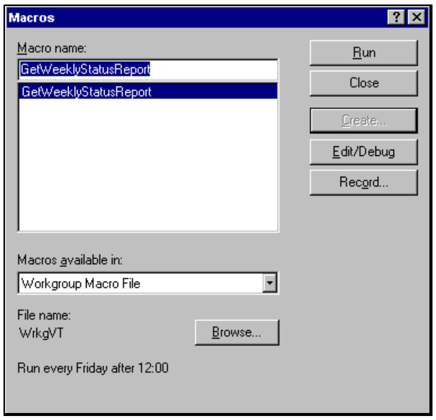
```
Sub GetWeeklyStatusReport
   '! Run every Friday after 12:00
```

```
   ' initiate the file transfer on the host
   Session.Send "SX Wstatus.TXT"

   ' initiate the reception of the file on the PC
   Transfer.ReceiveFile "Wstatus.TXT"
End Sub
```

Note White space (extra spaces, carriage returns, and tabs) that makes the macro more readable is ignored by the compiler.

When you open the Tools>Macros dialog and select the macro, the dialog looks like this:



Notice that the instructions that appear in the second line of the macro text (**`'! Run every Friday after 12:00`**) now appear below the name of the module in which the macro is stored.

Using SmarTerm's objects

An *object* is a special kind of programming construct that organizes related settings and tasks into a single, *object-oriented* model. This model provides a common syntax for all related tasks, whether they involve changing settings, sending commands, or communicating with other applications. A macro accomplishes all related tasks by accessing the *methods* (commands) and *properties* (settings) of the appropriate object.

The syntax for accessing the methods and properties of an object is quite simple: **`Object.Method`** or **`Object.Property`**. To assign the current setting of an object's property to a variable, you use **`Variable = Object.Property`**. To use an object's method, you use **`Object.Method`**.

For example, suppose that you want to create a macro that gets the version number of SmarTerm and then displays it in the SmarTerm window. In a *procedural* language you might need to use two macro commands that use completely different syntax, such as:

```
LatestVersion$ = Version$( )! Get version number
Send (LatestVersion$)! Display version number
```

With this kind of macro language you need to learn a new syntax each time a different programmer adds a new feature. The macro code is hardly self-explanatory (version of what? Send it where?), and of course the presence or absence of parentheses, arbitrary as it seems, will make or break the macro.

With the object-orientation of the macro language, the version number and the session window are considered part of the SmarTerm application object, so you can use one statement for both tasks:

```
Session.Echo Application.Version
! Display the version number in the session window
```

You will use this object-oriented approach to control SmarTerm from a macro. In addition, if you create your own data structures, you will access the members of those structures using the same object-oriented syntax.

Understanding the SmarTerm objects

There are SmarTerm objects corresponding to the tasks basic to host connection: **`Application`** (controlling SmarTerm), **`Session`** (communicating with the host), **`Circuit`** (connecting to the host), **`Transfer`** (transferring files), and **`Clipboard`** (moving information between SmarTerm and the Windows Clipboard). There are also objects that simplify the creation of a user interface (**`Msg`** and **`Dlg`**) and the handling of errors (**`Err`**). These are all briefly described in the following sections. All object properties and methods begin with the object name and are listed in alphabetical order in this manual and in the online help.

Application

The **`Application`** object is SmarTerm itself. With the **`Application`** object you control or have access to those properties of SmarTerm that are not session-dependent. You can also access methods that are not session-dependent.

Note The Application object should not be confused with the macro commands that begin App, such as AppActivate. The App commands provide access to external Windows and DOS applications, not to SmarTerm.

The Application object includes one sub-object, the Sessions collection. This sub-object gives you access to the set of sessions running or available to run at a given time. You access the properties and methods of all this Application sub-objects with a syntax very similar to that for the primary objects: **`Application.Sessions.Property`** or **`Application.Sessions.Method`**. For example, you can count the number of open session files with **`Application.Sessions.Count`**.

Session

With the **`Session`** object you control or have access to those properties of SmarTerm that are session-dependent. You can also access methods that are session-dependent.

The **`Session`** object includes five sub-objects that help you handle the flow of events that occurs between SmarTerm and the host.

You access the properties and methods of all of these Session sub-objects with a syntax very similar to that for the primary objects: **`Session.Object.Property`** or **`Session.Object.Method`**. For example, you set the keycode that SmarTerm should wait for with the **`Session.Keywait.Keycode`** property.

The primary documentation for the **`Session`** subobjects is in the online help system. The following sections briefly explain each subobject.

Collect The **`Session.Collect`** object allows you to pause the macro while it collects strings of text from the host. You can use the text you collect in any fashion you choose (but if you need to collect text and store it in a file, the **`Session.Capture`** or **`Session.Screentofile`** commands are more efficient). If you do not need to use the text sent by the host, but simply need to control the flow of the macro based on text sent from the host, consider using the **`Session.Stringwait`** subobject.

Note Since the **`Session.Collect`** object collects only text, it is not available if you are using a form-based session type, such as IBM 3270 or 5250. For form-based session types, use the **`Session.Eventwait`** object to wait for data from the host.

There are commands that allow you to start collecting text, indicate the signal to end collecting, and determine whether or not the collected text is passed on to the screen. There is one **`Session.Collect`** object per session. You can either trust SmarTerm to re-initialize all properties each time the object is used after the previous collection has finished, or you can use the **`Session.Collect.Reset`** command before each use of the **`Session.Collect`** object to clear all previous values of the object (such as the collected string or a timeout value).

Eventwait The **`Session.Eventwait`** object allows you to pause the macro while it checks to see if SmarTerm has sent one or more form pages to the host or received one or more form pages from the host. The **`Session.Eventwait`** object does not store the data on the pages sent to or received from the host.

Note Since the **`Session.Eventwait`** object only waits for form pages, it is not available if you are using a text-based session type, such as Digital VT, ANSI, SCO ANSI, or Wyse. For text-based session types, use the **`Session.Collect`** or **`Session.Stringwait`** object to wait for data from the host.

There are commands that allow you to start waiting for form events and indicate the signal to end waiting. There is one **`Session.Eventwait`** object per session. You can either trust SmarTerm to re-initialize all properties each time the object is used after a **`Session.Eventwait`** operation, or you can use the **`Session.Eventwait.Reset`** command before each use of the **`Session.Eventwait`** object to clear all previous values of the object (such as the number of pages to receive before resuming the macro).

Keywait The **`Session.Keywait`** object allows you to pause the macro while it checks for a keystroke or mousebutton press. You can have the macro check for any keystroke, for a specific keystroke, for a certain number of keystrokes of any kind, or for a specific mousebutton. You can also set a timeout value. There is one **`Session.Keywait`** object per session. You can either trust SmarTerm to re-initialize all properties each time the object is used after the previous **`Session.Keywait`** operation, or you can use the **`Session.Keywait.Reset`** command before each use of the **`Session.Keywait`** object to clear all previous values of the object.

Stringwait The **`Session.Stringwait`** object allows you to pause the macro while it checks for receipt of a string of text from the host. This object does not store the text received from the host, so if you need to use the text received from the host, use the **`Session.Collect`** object or the **`Session.Capture`** or **`Session.Screentofile`** command.

Note Since the **`Session.Stringwait`** object waits only for text, it is not available if you are using a form-based session type, such as IBM 3270 or 5250. For form-based session types, use the **`Session.Eventwait`** object to wait for data from the host.

There are commands that allow you to start waiting for a string, indicate whether to match the string exactly or not, set a maximum timeout and a maximum number of characters to wait through, and determine whether or not the string has been matched. There is one **`Session.Stringwait`** object per session. You can either trust SmarTerm to re-initialize all properties each time the object is used after the previous collection has finished, or you can use the **`Session.Stringwait.Reset`** command before each use of the **`Session.Stringwait`** object to clear all previous values of the object (such as the collected string or a timeout value).

Lockstep The **`Session.Lockstep`** object allows you to ensure that SmarTerm and the host remain in sync with each other while the macro is monitoring data sent to or received from the host. This prevents your macro from failing in situations where the host sends or receives data faster than SmarTerm can handle internally. For example, if you use the **`Session.Stringwait`** object to wait for a prompt from the host, it is possible that the host may send the string you are waiting for while SmarTerm is setting up the **`Session.Stringwait`** object. The wait will then fail, because the macro never sees the string even though the host has sent it. On the other hand, if you begin by setting up the **`Session.Lockstep`** object and then start waiting for the string, SmarTerm handles flow control with the host such that no characters are dropped.

`Session.Lockstep` is a simple enough object that there are only three methods for it: **`Start`**, **`Stop`**, and **`Reset`**.

Circuit

The **`Circuit`** object is the current communication method in use by the active session. With the **`Circuit`** object you control or have access to those properties of SmarTerm that relate to the details of host connection, such as any settings that appear on the Connection>Properties dialog (which vary depending on the communication method). You can also access methods that relate to the details of host connection (which also vary depending on the communication method).

All Circuit methods and properties unique to a given communication method are prefixed with the name of the communication method, such as Circuit.TelnetHostName. As of this version of SmarTerm, the supported communication methods are LAT, modem, serial, SNA, and Telnet.

Transfer

The **`Transfer`** object is the current transfer method in use by the active session. With the **`Transfer`** object you control or have access to those properties of SmarTerm that relate to file transfer, such as generic File menu commands and any settings that appear on the Properties>File Transfer Properties dialog (which vary depending on the transfer method). You can also access methods that relate to the details of host connection (which also vary depending on the transfer method).

Note For macro commands dealing with data capture from the host, see the methods and properties of the Session object.

All methods and properties unique to a given transfer method are prefixed with the name of the transfer method, such as Transfer.FTPHostName. As of this version of SmarTerm, the supported file transfer methods are FTP, IND$FILE, Kermit, XModem, YModem, and ZModem. However, because ZModem handles so many file transfer issues automatically, there are no unique Transfer properties or methods for it.

Clipboard

The **`Clipboard`** object is a special object that provides access to the Windows Clipboard, allowing you to transfer text between SmarTerm and another Windows application. With the **`Clipboard`** object you can cut and copy text from the session window to the clipboard, paste text into the session window from the clipboard, and clear the clipboard. You can also set the format of clipboard text and pipe text to and from the clipboard directly from a macro.

Msg

The **`Msg`** object provides a modeless dialog—that is, a dialog that the user must respond to before continuing. (The standard Windows File>Open dialog is a good example of a modeless dialog: you must click either Open or Cancel to dismiss the dialog.) SmarTerm's **`Msg`** object can contain text and a thermometer control in addition to an OK button and a Cancel button. Macro commands allow you to create, change the contents of, and close the dialog.

Dlg

The **`Dlg`** object provides easy access to dynamic dialogs defined in your macros. Each Dlg method works as either a statement or a function, allowing you to check return values or ignore them as you prefer. The use of the **`Dlg`** object and dialog procedures in general are described in more detail in "Using a Dynamic Dialog in a Macro" on page 79.

Err

The **`Err`** object allows you to create your own routines to handle errors returned by the compiler, OLE objects, and external DLLs. You can also construct macro code to raise errors as necessary. The methods and properties of the **`Err`** object provide access to the calling OLE object or external DLL, and the source if possible.

Modules and collectives

The locations where macros are stored (the macro *modules*) are primarily determined by settings stored in the session file. The modules available in a session, called the macro *collective*, do not share source code, but they can share variables with each other. Moreover, some members of the collective can act as repositories of shared macros available to all the other members of the collective. This allows you to create multiple session files that employ different sets of macros, but which may also share some macros. For example, you may always log onto one host in the same way, but run different applications at different times that require special macros. You can set up a session file for each host application that employs the same login macro, but loads a unique set of macros appropriate to each application. The session-based macro collective also allows you to share macros among many users simply by sharing the locations of certain modules (see “Possible improvements” on page 83).

A macro collective consists of:

- Macros stored in the User macro file
- Macros stored in the session file, including the Session_Connect macro, which runs when the session connects to the host; the Session_QueryClose macro that runs when the session is closed; and any SmartMouse event handlers
- Macros compiled and saved as files with the **`.PCD`** extension in the program folder (see “Compiling Macros” on page 90 for instructions).
- Macros stored in the currently running macro file loaded with the Other Macro file option on the Tools>Macros dialog
- Macros embedded in the currently loaded keyboard map
- Macros embedded in the currently loaded SmarTerm Buttons palette
- Macros embedded in the currently loaded HotSpots file

Global variables can be declared in any member of the collective and then accessed by any member of the collective. Subroutines and functions stored in the first three locations listed above (the User macro file, the session file, and any compiled macro files) are always available to each other and to any loaded tools (such as keyboard maps, Buttons, HotSpots, and the Other macro file). Subroutines and functions stored in loaded tools, however, are not accessible to other members of the collective.

Note You must use the Declare statement to prototype functions in the User macro file, session file, and compiled macro files that you want accessible to other members of the collective. This step is not required for subroutines unless you have also turned on Option Explicit to require prototyping of external routines. For clarity's sake, we recommend that you turn on Option Explicit and prototype all functions and subroutines. See "Declare" on page 209 and "Option Explicit" on page 365 for more information.

The user macro file is intended as a location where individuals can build up a collection of their own macros. By default, SmarTerm assumes that you will tend to organize macros based on session type, so the default user macro files assumed for a new session are:

Session Type	User Macro file
Digital VT, ANSI, SCO ANSI	USERVT.STM
Data General DASHER, Wyse	USERDG.STM
IBM 3270, IBM 5250	USERIBM.STM

You can select new user files for a given session with the Tools>Macros dialog or through Properties>Session Options>Macros tab. You can change the location where SmarTerm looks for macros through Properties>Options>File Locations tab. If you do so, be aware that you cannot make this change on a per-session basis; all sessions must store their user macros in common folders.

In a server installation of SmarTerm, the user macros folder can reside on each user's PC or the user folder on the network.

The last entry in the list above, Other Macro File, is a special case. This feature allows you to select any macro file, select a specific macro in it, and click Run to run the macro.

Predefined login and logout macros

As part of a session's macro collective, SmarTerm provides for two predefined macros: Session_Connect and Session_QueryClose macro. The Session_Connect macro runs automatically when the session file is opened, and the Session_QueryClose macro runs automatically when the session file is closed. These macros are stored in the session's STW file under the heading `[Script]`.

Session_Connect macro

There are a number of ways in which you can create the Session_Connect macro. One way is to use the Tools>Macros dialog to write it from scratch; another way is to record an actual login when you create the session (you can always edit the resulting macro to add more commands). If you record a login, clicking Stop on the macro recorder toolbar after you enter your password, you get a skeletal login macro that looks something like this:

```
Sub Session_Connect
    '! This macro is run automatically when the session opens.

    Dim nContinue as Integer
    Dim nTimeOut as Integer

    ' The default timeout for each command is 3 minutes.
    ' Increase this value if your host requires more time
```

```
    ' for each command.
    nTimeOut = 180

    Dim LockStep As Object
    Set LockStep = Session.LockStep
    LockStep.Start

    While (Circuit.Connected = False)
    Wend

    ' Wait for response from host.
    Session.StringWait.Timeout = nTimeout
    Session.StringWait.MatchStringExact "Username: "
    if Session.StringWait.Start = smlWAITTIMEOUT then
        nContinue = QuerySyncError()
        if nContinue <> ebYes then End
    end if

    Session.Send "nguyenp" + chr(13)

    ' Wait for response from host.
    Session.StringWait.Timeout = nTimeout
    Session.StringWait.MatchStringExact "Password: "
    if Session.StringWait.Start = smlWAITTIMEOUT then
        nContinue = QuerySyncError()
        if nContinue <> ebYes then End
    end if

    Session_Connect_PasswordHandler 1
    Session.Send chr(13)

    Set LockStep = Nothing

End Sub
```

Everything in this sample Session_Connect macro was generated automatically by SmarTerm, with the exception of the account name (**nguyenp**), which was entered by the person logging onto the host. Let's look briefly at each section of the macro.

The macro begins with a description line explaining when the macro runs, which will appear at the bottom of the Tools>Macros dialog when the Session_Connect macro is selected. This is followed by the definition of several variables and the assignment of values to those variables:

```
Dim nContinue as Integer
Dim nTimeOut as Integer

' The default timeout for each command is 3 minutes.
' Increase this value if your host requires more time
' for each command.
nTimeOut = 180

Dim LockStep As Object
Set LockStep = Session.LockStep
LockStep.Start
```

Dim (short for Dimension) is the standard BASIC command to define a variable. Notice that the macro uses the **as <Type>** notation to select a data type for each variable (as in **Dim nContinue as Integer**).

This is the clearest way to define a variable's type, but you can also use the type-definition character at the name to shorten the command (as in **`Dim nContinue%`**).

The variable **`nContinue`**, which is used to determine if there has been an error in the login, is assigned a value later in the macro.

The variable **`nTimeOut`**, which is used to halt the macro if there is no response from the host, is assigned the value 180 using the simple assignment statement **`nTimeOut = 180`**, although the macro could have used the wordier **`Let nTimeOut = 180`** method. As the comment preceding the assignment statement indicates, a value of 180 equals three minutes, so this macro will wait three minutes for the host to respond before automatically stopping. (Because this variable is used by the SmarTerm **`Session.Stringwait`** object later in the macro, its value must be specified in seconds). This is the default setting only. You can always edit the Session_Connect macro to shorten or lengthen the timeout just by changing the value assigned to **`nTimeOut`** in this statement.

The next three commands define a variable of type **`object`**, assign that variable to the SmarTerm **`Session.Lockstep`** object, and then send the Start command to that object. (For more about objects, see “Using SmarTerm’s objects” on page 27.) The Session_Connect macro sets up a **`Session.Lockstep`** object to ensure that SmarTerm and the host stay in sync with each other, so that SmarTerm always waits for complete responses from the host before running the next macro commands. You do not have to use this object to maintain synchrony, but it is by far the easiest way.

Next, the macro sets up a short **`While`** loop to wait for the initial host connection:

```
While (Circuit.Connected = False)
Wend
```

This command uses the SmarTerm **`Circuit`** object to test whether or not the initial host connection has been made. (Again, SmarTerm objects are described in detail later in this chapter). This is done by comparing the value of **`Circuit.Connected`** with the built-in constant **`False`**. As long as **`Circuit.Connected = False`**, the initial connection has not been made and SmarTerm will just keep making the comparison.

As soon as the connection has been made, SmarTerm sets **`Circuit.Connected`** to **`True`** and the **`While`** loop ends. Notice that SmarTerm did not set a timeout for this loop. The initial host connection is handled by the low-level drivers for the communication method, so the timeout cannot be changed by the application.

Once the connection has been made, SmarTerm begins the section of the macro that handles the actual login to the host. First the macro waits to get the **`Username`** prompt from the host (which it simply read off the screen when the macro was recorded):

```
' Wait for response from host.
Session.StringWait.Timeout = nTimeout
Session.StringWait.MatchStringExact "Username: "
if Session.StringWait.Start = smlWAITTIMEOUT then
```

```
        nContinue = QuerySyncError()
        if nContinue <> ebYes then End
   end if
```

This block first sets the length of time SmarTerm will wait for the **Username** prompt from the host by setting the **Timeout** property of the SmarTerm **Session.StringWait** object to the value stored in **nTimeout** earlier in the macro (180 seconds). Then it tells SmarTerm what host string to wait for by sending the **MatchStringExact "Username: "** message to the SmarTerm **Session.StringWait** object.

Finally, the macro sets up an **If** loop to determine whether or not the host has sent the **Username** prompt. If SmarTerm receives the **Username** prompt before the timeout expires, then the macro skips the If loop and proceeds to the next section of the macro. If the timeout has expired, a messagebox appears that indicates an out-of-sync error and asks if the user wants to continue (this error handler, the QuerySyncError function, is defined as a separate subroutine after the end of the Session_Connect subroutine). If the user clicks No, then the macro ends; if Yes, then the macro continues even though it probably won't work anymore. This function is self-explanatory, so we will not go into it here.

If SmarTerm has received the **Username** prompt, it then sends the username typed in when the macro was recorded, and then waits for the host to prompt for the password:

```
     Session.Send "nguyenp" + chr(13)

     ' Wait for response from host.
     Session.StringWait.Timeout = nTimeout
     Session.StringWait.MatchStringExact "Password: "
     if Session.StringWait.Start = smlWAITTIMEOUT then
         nContinue = QuerySyncError()
         if nContinue <> ebYes then End
    end if
```

The macro sends the username by sending the **Send** message to the SmarTerm **Session** object. The complete username is constructed as **"nguyenp" + chr(13)**, which is the text typed by the user concatenated with a carriage return (character 13 in the standard ASCII table). The loop that waits for the password is exactly the same as the one that waits for the username, except that now the string the macro waits for is **"Password: "**.

When SmarTerm receives the password, it calls the **Session_Connect_PasswordHandler** function, which is defined at the bottom of the Session_Connect macro module. The call looks like this:

```
Session_Connect_PasswordHandler 1
Session.Send chr(13)
```

The actual **Session_Connect_PasswordHandler** subroutine differs from macro to macro depending on whether you chose to save the Session_Connect macro in a secured or unsecured way. If you chose secured, then the subroutine looks something like this:

```
Sub Session_Connect_PasswordHandler(i as Integer)
' This procedure is called to send a password to the host.
'
' You have chosen not to store passwords in your macro file, so
' this_ procedure prompts for a password.
```

```
        ' Wait for user to enter the password.
    Session.Send AskPassword$("Enter password:")
End Sub
```

This version of the subroutine displays a messagebox asking the user for a password. The user then types in the password, which is displayed as a series of asterisks (*) in the dialog, then clicks OK (this is the AskPassword$ function). The macro then uses **`Session.Send`** to send the password to the host. There is no error handling at this point, however, so if the user types an incorrect password it's up to the host to deal with it.

If you chose to save the macro unsecured, the **`Session_Connect_PasswordHandler`** subroutine looks something like this:

```
Sub Session_Connect_PasswordHandler(i as Integer)
' This procedure is called to send a password to the host.
' You have chosen to store passwords in your macro file, so this
' procedure simply sends the correct password.

    select case i
        case 1
            Session.Send "chaothay"
    end select

End Sub
```

In this case, as the comment observes, the macro simply sends the text you typed in when recording the macro.

The final line of the Session_Connect macro deals with the **`Session.Lockstep`** object created at the very beginning of the macro:

```
Set LockStep = Nothing
```

This line destroys the **`Session.Lockstep`** object. This is important because, as the section in this chapter on SmarTerm objects explains, you can have only one **`Session.Lockstep`** object per session. Destroying the object as soon as you are finished using it ensures that the next time you need to maintain synchrony between SmarTerm and the host there will be no residual data that might confuse the situation.

Session_QueryClose macro

The Session_QueryClose session macro is a logout macro – a counterpart to Session_Connect. Its purpose is to make it easy to customize SmarTerm behavior when an attempt is made to close a session. For example, a system administrator could write a macro that reads the screen and verifies that the user has just entered a logout command. If the user hasn't, this macro could emit a warning message, to remind the user to exit any host applications first, and then logout properly.

This macro can be written to test for certain conditions and affect the session close operation accordingly, even canceling the close attempt altogether.

Below is an example of this macro as an empty shell, to illustrate its parameters:

```
Sub Session_QueryClose
   ....
   [statements go here]
   ....
End Sub
```

Why macros, modules, and collectives

Although the macro-module-collective system may seem confusing at first, it can provide major benefits in *interoperation*. That is to say, all of the macros in all of the modules participating in the collective can share subroutines and data with each other. This allows you to reuse macros rather than rewrite them, and lets you create more complex macros that interact with each other to produce more sophisticated results.

Note The module called Other Macro File in the Tools>Macros dialog is a special case. This module, while fully participating in the collective whenever one of its macros is running, withdraws from the collective when its macros are not running. Macros that must participate in the collective at all times should be placed in the user macro file.

To get a better idea of how this interoperation works, let’s consider an example. Suppose that you want these steps to occur:

1. When you log onto the host, the Session_Connect macro sends your user name and password to the host.
2. The host sends a line of text displaying a “virtual circuit number” corresponding to your connection.
3. Your login macro records the virtual circuit number (which must be supplied as a parameter to the print spooler later on in the session) and stores it where a SmarTerm button macro can access it. This requires a *public* or *global* variable – a variable whose value can be read and written by more than one macro in the collective.
4. A SmarTerm-button macro later gets the saved virtual circuit number and uses it in a print spooler command sent to the host.

What follows is a simple example of this interoperation that assumes that you are not taking advantage of macros. We can expand this example to show the power of shared macros in the collective (see “Possible improvements” on page 83).

This example requires interoperation between two macros in the collective, the Session_Connect macro and a macro embedded in a SmarTerm button. First let's look at the Session_Connect macro. There are a number of ways in which you can create this macro. One way is to use the Tools>Macros dialog to write it from scratch; another way is to record an actual login when you create the session and then modify that recorded Session_Connect macro. If you record a login, you get the login macro that we discussed earlier in this chapter.

At the top of the Session_Connect macro module, we define a public variable named **`VirtualCircuit`** as follows:

```
Public VirtualCircuit as String

Sub Session_Connect
    '! This macro is run automatically when the session opens.
.
.
.
End Sub
```

The keyword **`Public`** identifies the variable as one available to all modules in the collective. This keyword is actually optional; you could use **`Dim`** instead, and the macro compiler will assume that you wanted the variable to be public. If you need a variable to be shared between macros in one module, but invisible to macros in other modules in the collective, use the keyword **`Private`** instead.

Having defined **`VirtualCircuit`** as a public variable, we then set up the macro commands that read the virtual circuit number off the screen. These commands go inside the Session_Connect macro since right after logon is the only time that the host displays this information. However, the commands should go before the command that destroys the **`Session.Lockstep`** object so that we can be sure that SmarTerm and the host are in sync.

```
Sub Session_Connect
.
.
.
    Session_Connect_PasswordHandler 1
    Session.Send chr(13)

    ' Wait for response from host.
    Session.StringWait.Timeout = nTimeout
    Session.StringWait.MatchStringExact "Circuit Number: "
    if Session.StringWait.Start = smlWAITTIMEOUT then
        nContinue = QuerySyncError()
        if nContinue <> ebYes then End
    end if

    ' Read circuit number from screen. We assume a single digit.
    Session.Collect.MaxCharacterCount = 1
    Session.Collect.Start

    ' Now set VirtualCircuit to the number collected from host.
    VirtualCircuit = Session.Collect.CollectedCharacters

    Set LockStep = Nothing

End Sub
```

This block of commands is really quite simple. First, we wait for the prompt **`"Circuit Number: "`** exactly as we waited for the username and password prompts. Then we read a single digit from the host using the SmarTerm object **`Session.Collect`**.

```
    ' Read circuit number from screen. We assume a single digit.
    Session.Collect.MaxCharacterCount = 1
    Session.Collect.Start
```

The **`Session.Collect`** object automatically stores a single character in the property **`Session.Collect.Collected`**. Therefore, all we need to do to use the digit obtained is store it in the public variable **`VirtualCircuit`**:

```
    ' Now set VirtualCircuit to the number collected from host.
    VirtualCircuit = Session.Collect.CollectedString
```

Now whenever you open this session and connect to the host, the Session_Connect macro always creates a public variable called **`VirtualCircuit`** and stores the virtual circuit number obtained from the host in it. That variable and the number stored in it are now available to all macros in the collective. The only catch is that each module that needs to use a public variable declared in a different module must also declare it as a public variable. For example, if you create a SmarTerm button that starts a print spooler, sending the virtual circuit number obtained by the Session_Connect macro, the following statement must appear at the top of the SmarTerm button macro's module. Then the print spooler macro can send the number in the variable to the host print spooler:

```
Public VirtualCircuit as Integer

Sub CallPrintSpooler
    ! This macro runs the print spooler.
.
.
.
    Session.Send ViritualCircuit
.
.
.
End Sub
```

Programming Macros

This chapter describes how to:

- Use the Macro Editor
- Create the user interface for a macro
- Use SmarTerm objects
- Communicate with a host via macros
- Create compiled macro files

Using the macro editor

This section explains how to use the macro editor, a tool that enables you to edit and debug macros. It begins with some general information about working with the Macro Editor and then discusses editing your macros, running your macros to make sure they work properly, debugging them if necessary, and exiting from the Macro Editor.

The macro editor window

To edit a macro, select Tools>Macros to see the macros dialog. Then either select an existing macro file and macro and click Edit/Debug, or just enter a macro name and click Create to start editing a new macro. The macro editor window then appears. It contains the following elements:

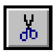
- **Toolbar** with buttons for controlling the macro editor
- **Edit pane** that contains the macro you are editing
- **Status bar** that displays the current location of the insertion point
- **Watch pane** that allows you to monitor the values of variables

Getting help

You can get online help for the macro editor and use of the macro language using the standard Windows methods. In addition, you can get specific help on a keyword or a watch variable by placing the insertion point within the text you have a question about and pressing F1.

Using the toolbar

The following list summarizes the buttons on the macro editor toolbar, which provide quick access to the menu commands.



Edit>Cut

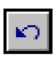
Cuts the selected text to the Clipboard.

Edit>Copy

Copies the selected text to the Clipboard.

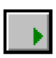
Edit>Paste

Pastes the contents of the Clipboard into the macro.



Edit>Undo

Undoes the last edit. Click multiple times to undo multiple edits.

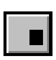


Macro>Start

Runs the macro.

Break

Pauses the macro and points to the next line to be executed.



Macro>Stop

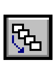
Stops running the macro.

Debug>Toggle Breakpoint

Adds or removes a breakpoint.

Debug>Add Watch

Opens the Add watch dialog.



Calls

Lists the procedures called by the macro. Available only when a running macro is paused.

Debug>Single Step

Executes the next line of a macro and then pauses. If the macro calls another macro procedure, execution continues into each line of the called procedure.

Debug>Procedure Step

Executes the next line of a macro and then pauses. If the macro calls another macro procedure, the compiler runs the called procedure in its entirety.

Using accelerators

The macro editor supports the Microsoft Office standard for common editing functions (such as Ctrl+C and Ctrl+Insert to copy selected text to the clipboard). In addition, the macro editor provides the following accelerator keys for commonly used commands.

Key(s)	Commands
Ctrl+A	Edit>Select All: Selects all text in the module.
Ctrl+Break	Break (Pause).
Ctrl+F	Edit>Find: Opens the Find dialog.
Ctrl+G (F4)	Edit>Goto Line: Opens the Goto Line dialog.
Ctrl+K	Macro>Check syntax.
Ctrl+Y	Yank: Deletes the entire line containing the insertion.
Home	Moves the insertion point to the beginning of the line.
Ctrl+Home	Moves the insertion point to the beginning of the module.
PgDn	Moves the insertion point down one windowful.
Ctrl+PgDn	Moves the insertion point right one windowful.
PgUp	Moves the insertion point up one windowful.
Ctrl+PgUp	Moves the insertion point left one windowful.
Ctrl+Left arrow	Moves the insertion point one word left.
Ctrl+Right arrow	Moves the insertion point one word right.
End	Moves the insertion point to the end of the line.
Ctrl+End	Moves the insertion point to the end of the module.
Shift+navigation key	Move the insertion point, selecting the intervening text. For example, Shift+Ctrl+Left arrow selects the word to the left of the insertion point.
Esc	Deactivates the Help pointer if it is active. Otherwise, exits your macro and returns you to the Tools>Macros dialog.
F2	During debugging, opens the Modify Variable dialog for the selected watch variable in the watch pane. You can also double-click the variable.
F3	Edit>Find Next.

Key(s)	Commands
F5	Macro>Run.
F6	Switches between the watch pane and the edit pane.
F8	Debug>Single Step.
Shift+F8	Debug>Procedure Step.
F9	Debug>Toggle breakpoint.
Shift+F9	Debug>Add watch.

Editing macros

In most respects, editing macro code with the macro editor is like editing regular text with a word-processing program. However, the macro editor also has certain capabilities specifically designed to help you edit macro code.

In this section you'll learn how to move around within macros, select and edit text, add comments, break long macro statements across multiple lines, search for and replace text, and check the syntax.

Moving around in a macro

Like all text editors, the macro editor lets you move around in a macro with the cursor keys and the mouse. However, the macro editor differs from most word-processing programs in that it allows you to place the insertion point anywhere within your macro, including "empty space," such as a tab's expanded space or the area beyond the last character on a line. This feature allows you to place comments anywhere in the macro file, so that you can place comments next to the relevant lines in the macro. A corollary to this feature is that there is no automatic wordwrap in the macro editor.

In addition, there are several special movement commands. You can jump to:

- The start or end of the line with the Home and End keys.
- Any line in the macro file by selecting Edit>Goto line (Ctrl+G or F4) and typing in a line number. This is particularly helpful if you receive a runtime error message that specifies the number of the line containing the error.
- Up or down by windowfuls with PageUp and PageDown, and left or right by windowfuls with Ctrl+PageUp and Ctrl+PageDown.
- To the top or bottom of the file containing the macro with Ctrl+Home and Ctrl+End. (Remember, multiple macros can be stored in one macro file).

Color coding in macros

When you enter certain types of text in the macro editor, the text automatically appears in a distinctive color. The default colors, which you can change, are:

- Blue for keywords

- Black for normal text
- Green for comments
- Red for breakpoints

Adding comments to macros

Comments are lines or portions of lines of macro code that are ignored when a macro runs. You can add comments to macros to remind yourself or others of how your code works or to temporarily disable blocks of code.

Comments are indicated with the keyword REM or with a single apostrophe ('), which causes the compiler to ignore all following text until the next line. You can thus have a full-line comment by beginning a line with REM or an apostrophe, or you can follow executable code with a comment on the same line just by inserting **`:REM`** (the colon is required) or an apostrophe at the point where you want the comment. Just remember that, although you can place a comment at the end of a line containing executable code, you cannot place executable code at the end of a line containing a comment.

You can also use C-style multiline comment blocks **`/*`**...**`*/`**, as follows:

```
Session.Echo "Before comment"
/* This stuff is all commented out.
This line, too, will be ignored.
This is the last line of the comment. */
Session.Echo "After comment"
```

C-style comments can be nested.

Breaking a macro statement across multiple lines

By default, a single macro statement can extend only as far as the right margin, and each new line constitutes a new statement. However, you can break a long statement across two or more lines with the *line-continuation character*, the underscore (_). Any line that ends with a space followed by the underscore character is combined with the next line and compiled as a unit.

For the most part, long lines stitched together with underscores indicate weak design, and should be avoided.

Searching and replacing

The macro editor makes it easy to search for specified text in your macro and automatically replace instances of specified text. The Edit>Find command (Ctrl+F), Edit>Find Next command (F3), and Edit>Replace command all work as you would expect in a text editor.

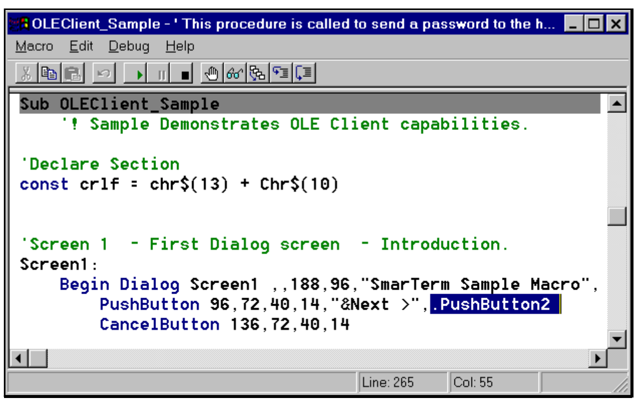
Checking the syntax of macros

When you try to run or debug a macro whose syntax hasn't been checked, the Macro Editor first performs a syntax check automatically. You can also check the syntax of a macro whenever you please with the Macro>Check syntax command (Ctrl+K). When you use this command, the macro editor checks the syntax of the entire macro, stopping the check when it finds the first syntax error (if there are any) and highlighting the line containing the error. You must correct the syntax error the macro editor found before continuing to check the syntax or running the macro.

Debugging macros

This section explains how to use the macro debugger integrated with the macro editor to find and correct errors in your macros. While debugging, you are actually executing the code in your macro line by line. Therefore, to prevent any modifications to your macro while it is being run, the edit pane is read-only during the debugging process. You are free to move the insertion point throughout the macro, select text and copy it to the Clipboard, set breakpoints, and add and remove watch variables, but you cannot make any changes to the macro code until you stop running it.

To let you follow and control the debugging process, the Macro Editor displays an *instruction pointer* on the line of code that is about to be executed—that is, the line that will be executed next if you either proceed with the debugging process or run your macro at full speed. When the instruction pointer is on a line of code, the text on that line appears in black on a gray background that spans the line. In the following illustration, the line beginning with the keyword **`Sub`** is marked with the instruction pointer. As a comparison, the block of text that says **`.PushButton2`** is shown with the highlighting used to indicate selected text.



Tracing macro execution

The Macro Editor gives you two ways to trace macro execution—single step and procedure step—both of which involve stepping through your macro code line by line. Single step simply traces through every line in the macro, going into each subroutine called by the macro in complete detail. Procedure step traces line by line through the code for the macro itself, but runs all of the subroutines called by the macro without showing the line-by-line detail. Single step is good for debugging relatively simple macros that do not call very many subroutines. Use procedure step on macros that call subroutines you have already debugged and do not need to see traced in detail.

Note Single-step doesn't work when a macro uses the SmarTerm Session.StringWait, Session.Collect, or Session.EventWait objects to control the timing and flow of the macro. In such macros you must use breakpoints instead.

To trace a macro:

1. Click the Single Step or Procedure Step button on the toolbar, or Press F8 (Single Step) or Shift+F8 (Procedure Step). The macro editor places the instruction pointer on the first line of the macro.

Note When you start a trace, there may be a slight pause before the trace actually begins while the macro editor compiles your macro. If it finds errors during compilation, you will have to correct them before you can continue debugging.

2. Repeat step 1 to run the marked line and then advance the instruction pointer to the next instruction. Each time you repeat step 1, the macro editor runs the line containing the instruction pointer and then moves to the next line.
3. When you finish tracing the macro, either select Macro>Start (F5 or the toolbar button) to run the rest of the macro at full speed, or select Macro>End (or the toolbar button) to stop running the macro.

While you are stepping through a subroutine, you may need to determine the subroutine calls by which you arrived at that point in the macro. You can do this with the Calls dialog.

To use the Calls dialog:

1. Click the Calls button on the toolbar. The Calls dialog appears, which lists the subroutine calls made by your macro in the course of arriving at the current subroutine.
2. To view one of the subroutines listed in the Calls dialog, highlight it and click Show. The macro editor then displays that subroutine, highlighting the currently running line. (Note, however, that the instruction pointer remains in its original location in the subroutine.)

When you are stepping through a subroutine, you may want to repeat or skip execution of a section of code. You can use the Set Next Statement command to move the instruction pointer to a specific line within that subroutine.

Note You can only use the Set Next Statement command to move the instruction pointer within the same subroutine.

To move the instruction pointer to another line within a subroutine:

1. Place the insertion point in the line where you want to resume stepping through the macro.
2. Select Debug>Set Next Statement. The instruction pointer moves to the line you selected, and you can resume stepping through your macro from there.

Setting and removing breakpoints

If you are debugging a long, complicated macro, stepping through it line by line can be quite time-consuming. An alternate strategy is to set one or more *breakpoints* at selected lines in your macro. Then, when you run the macro, it automatically pauses at each breakpoint, allowing you to examine the code or step through the lines only where necessary

You can set breakpoints anywhere in a macro, but only breakpoints on lines that contain macro commands, including lines in functions and subroutines are considered valid. (The macro editor beeps if you set an invalid breakpoint.) When you compile and run the macro, invalid breakpoints are automatically removed.

You can set breakpoints at any time while editing a macro or when a running macro has been paused. For example, if you know that there are certain sections you want to debug, you can set all of the breakpoints in the editor, and then run the macro to check the code at each breakpoint. Or, if the macro doesn't seem to be working properly, you can use the Break command (Ctrl+Break) to pause the macro, set a breakpoint, and then resume running the macro to move at full speed to the breakpoint.

To set a breakpoint:

1. Place the insertion point in the line where you want to start debugging.
2. Select Debug>Toggle Breakpoint (F9 or the Toggle Breakpoint button).

Note You can set up to 255 breakpoints in a macro.

Invalid breakpoints are removed automatically when the macro is compiled and run. When you exit the macro editor, all other breakpoints are also removed. You can also remove breakpoints manually.

To remove a single breakpoint:

1. Place the insertion point on the line containing the breakpoint that you want to remove.
2. Select Debug>Toggle Breakpoint (F9 or the Toggle Breakpoint button).

To remove all breakpoints:

Exit the macro editor or select Debug>Clear All Breakpoints.

Using Watch variables

As you debug your macro, you can use the *watch pane* to monitor selected variables. For each variable you select, the watch pane displays its context, name, and value. The values of the variables on the watch list are updated each time you pause the macro with a breakpoint or with the Break command (Ctrl+Break).

The Macro Editor permits you to monitor variables of fundamental data types, such as **`Integer`**, **`Long`**, **`Variant`**, and so on; you cannot watch complex variables, such as user-defined types or arrays, or expressions using arithmetic operators. You can, however, watch individual elements of user-defined types or arrays using the following syntax:

`[variable [(index,_)] [.member [(index,_)]]_]`

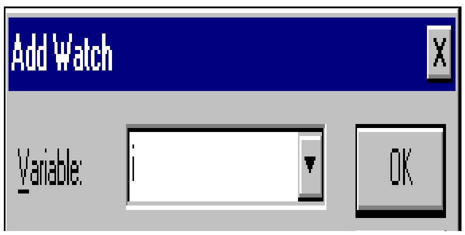
where ***`variable`*** is the name of the user-defined type or array variable, ***`index`*** is a literal number, and ***`member`*** is the name of a member of the user-defined type.

For example, the following are valid watch expressions:

Watch Variable	Description
`a(1)`	Element `1` of array **`a`**
`person.age`	Member ***`age`*** of the user-defined type **`person`**
`company`*(10,23)*.person.age`	Member ***`age`*** of user-defined type ***`person`*** that is at element 10,23 within the array of user-defined types called ***`company`***

To add a watch variable:

1. It is most flexible to add watch variables when running the macro, so begin by select Macro>Start (F5 or the Start button), then press Ctrl-Break to pause the macro. Or, insert a breakpoint at an appropriate location in the macro and then run it.
2. When the macro pauses, select Debug>Add Watch (Shift+F9 or the Add Watch button). The Add Watch dialog appears.

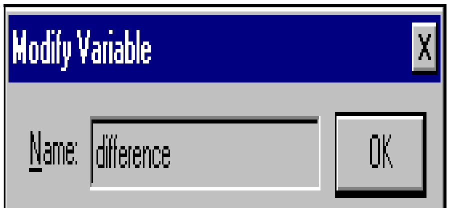


3. In the Procedure box, select the name of the procedure containing the variable you want to watch. If the variable you want to watch is global to the module, select "(All Procedures)".
4. In the Variable box, select the name of the variable you want to add to the watch variable list.
5. In the Script box, type or select the name of the macro containing the variable you want to watch. If you're creating a new name, don't include any spaces. If the variable you want to watch is global to the collective, select "(All Scripts)".
6. Click OK to add the variable to the watch variable list.

   The context, name, and value of the variable appear in a three-column list in the watch pane at the top of the macro editor window, along with any other variables you may have added during this editing session.

To modify the value of a watch variable:

1. Highlight the variable in the watch pane and select Debug>Modify Watch (F2), or just double-click the variable in the watch pane. The Modify Variable dialog appears.



2. Enter the new value for the variable in the Value field.
3. Click OK. The new value of your variable appears on the watch variable list.

   When you change the value of a variable, the macro editor converts the value you enter to match the type of the variable. For example, if you change the value of an **`Integer`** variable to 1.7, the macro editor converts this value from a floating-point number to an **`Integer`**, assigning the value 2 to the variable.

   When you modify a **`Variant`** variable, the macro editor determines both the type and value of your entry using the following rules (in this order):

If the new value is	Then
`Null`	The **`Variant`** variable is assigned **`Null`** (**`VarType 1`**).
`Empty`	The **`Variant`** variable is assigned **`Empty`** (**`VarType 0`**).
`True`	The **`Variant`** variable is assigned **`True`** (**`VarType 11`**).
`False`	The **`Variant`** variable is assigned **`False`** (**`VarType 11`**).
`number`	The **`Variant`** variable is assigned the value of ***`number`***. The type of the variant is the smallest data type that fully represents that number. You can force the data type of the variable by using a type-declaration letter following ***`number`***, such as **`%`**, **`#`**, **`&`**, **`!`**, or **`@`**.
`date`	The **`Variant`** variable is assigned the value of the new date (**`VarType 7`**).
Anything else	The **`Variant`** variable is assigned a **`String`** (**`VarType 8`**).

   The Macro Editor will not assign a new value if it cannot be converted to the same type as the specified variable.

To delete a watch variable:

1. Highlight the variable on the watch list.
2. Select Debug>Delete Watch or press the Delete key.

Creating Dialogs

Dialogs are created in two steps. First you define a *dialog template* that contains the definitions of the types, sizes, placement, and so forth of all the elements of a dialog. Then you use macro commands to create an *instance* of that dialog using the template you defined earlier in the macro.

To insert a new dialog template:

1. Place the insertion point where you want the new dialog template to appear in your macro. Bear in mind that the scope rules outlined above for variables and subroutines apply to dialog templates as well. If you want a dialog template to be available to all subroutines in a given macro file, define the template at the top of the file. If you want the template to be private to a specific subroutine, define it within that subroutine.
2. Select Edit>Insert New Dialog. The dialog editor appears, displaying a new dialog in its window.
3. Use the dialog editor to create the dialog.
4. Exit from the dialog editor and return to the macro editor.

   The Macro Editor automatically places the new dialog template generated by Dialog Editor in your macro at the location of the insertion point.

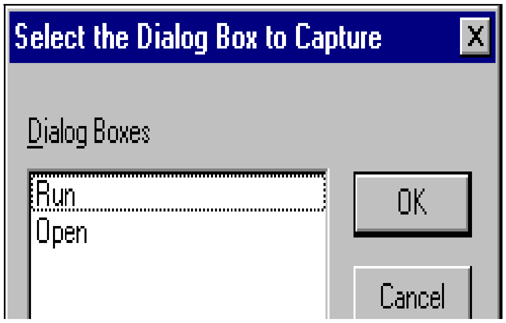
To edit an existing dialog template:

1. Select the lines of code that define the entire dialog template.
2. Select Edit>Edit Dialog. The dialog editor appears, displaying a dialog created from the code you selected.
3. Use the dialog editor to modify your dialog.
4. Exit from the dialog editor and return to the macro editor. The macro editor automatically replaces the dialog template you originally selected with the revised template generated by Dialog Editor.

To capture a dialog from another application:

You can capture the standard Windows controls from any standard Windows dialog in another application and insert those controls into the Dialog Editor for editing. Follow these steps:

1. Display the dialog you want to capture.
2. Open the Dialog Editor.
3. Select File>Capture Dialog. The Dialog Editor displays a dialog that lists all open dialogs that it is able to capture:



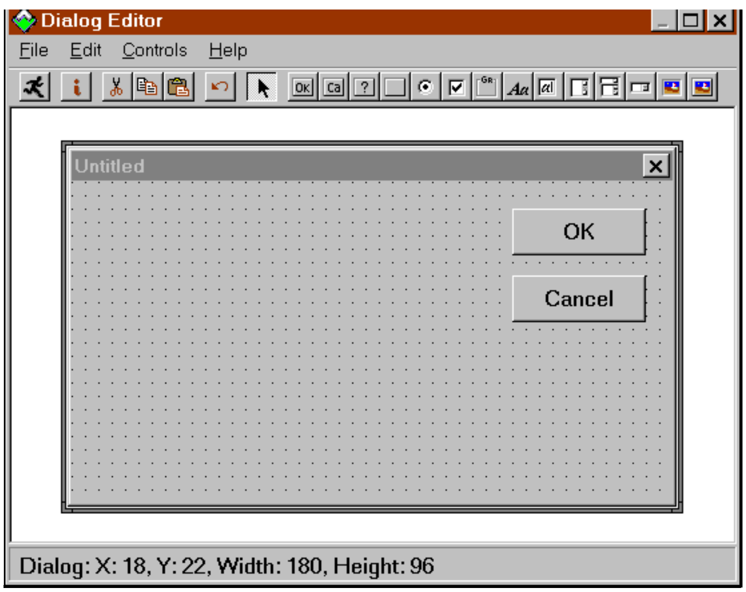
4. Select the dialog you want to capture, then click OK. The Dialog Editor now displays the standard Windows controls from the target dialog.

Note The Dialog Editor only supports standard Windows controls and standard Windows dialogs. You cannot capture custom dialogs or custom dialog controls.

Using the Dialog Editor

This section presents general information that will help you work most effectively with the Dialog Editor. It includes an overview of the Dialog Editor as well as a list of accelerators and information on using the Help system.

Before you begin creating a new custom dialog, the Dialog Editor looks like this:



The application window contains the following elements:

Toolbar

A collection of buttons that you can use to provide instructions to the Dialog Editor, as discussed in the following subsection.

Dialog

The visual layout of the dialog that you are currently creating or editing.

Status bar

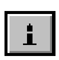
Provides key information about the operation you are currently performing, including the name of the currently selected control or dialog, together with its position on the display and its dimensions; the name of a control you are about to add to the dialog with the mouse pointer, together with the pointer's position on the display; the function of the currently selected menu command; and the activation of the Dialog Editor's testing or capturing functions.

Note Dialogs created with the Dialog Editor normally appear in an 8 point Helvetica font, both in the Dialog Editor's application window and when the corresponding macro code is run.

The Dialog Editor

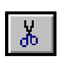
Test Dialog

Runs the dialog for testing.



Information

Displays information for the selected control.



Cut

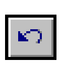
Removes the selected control from the dialog.

Copy

Copies the selected control to the clipboard.

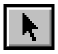
Paste

Inserts the clipboard into the active dialog.



Undo

Reverses the effect of the preceding editing change(s).

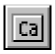


Select

Lets you select, move, and resize items and control the insertion point.

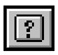
OK Button

Adds an OK button to your dialog.

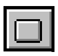


Cancel Button

Adds a Cancel button to your dialog.

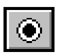


Help Button



Adds a Help button to your dialog.**Push Button**

Adds a push button to your dialog.

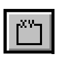


Option Button

Adds an option button to your dialog.

Check Box

Adds a checkbox to your dialog.

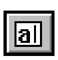


Group Box

Adds a group box to your dialog.

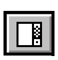
Text

Adds a text control to your dialog.



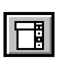
Text Box

Adds a text box to your dialog.



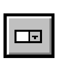
Listbox

Adds a listbox to your dialog.



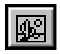
Combo Box

Adds a combo box to your dialog.



Drop List Box

Adds a drop-down listbox to your dialog.



Picture

Adds a picture to your dialog.

Picture Button

Adds a picture button to your dialog.

For more information, select Help.

Accelerators for the Dialog Editor

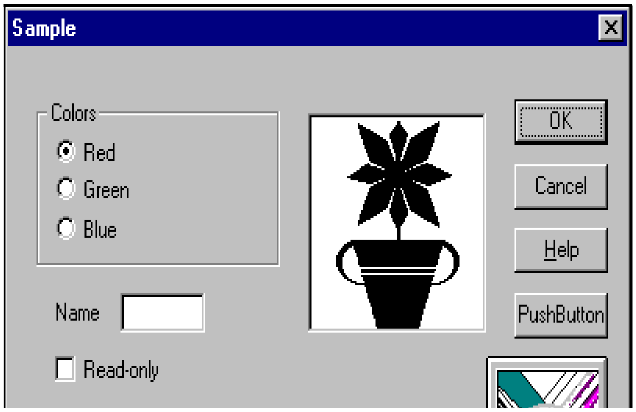
Key(s)	Function
Alt+F4	Closes the Dialog Editor.
Ctrl+C	Copies the selected dialog or control and places it on the Clipboard.
Ctrl+D	Creates a duplicate of the selected control.
Ctrl+G	Displays the Grid dialog.
Ctrl+I	Displays the Information dialog for the selected dialog or control.
Ctrl+V	Inserts the contents of the Clipboard into the Dialog Editor. If the Clipboard contains macro statements describing one or more controls, then the Dialog Editor adds those controls to the current dialog. If the Clipboard contains the template for an entire dialog, then the Dialog Editor creates a new dialog from the statements in the template.
Ctrl+X	Removes the selected dialog or control and places it on the Clipboard.
Ctrl+Z	Undoes the preceding operation.
Del	Removes the selected dialog or control.
F1	Displays Help for the active window.
F2	Sizes certain controls to fit the text they contain.
F5	Runs the dialog for testing.
Shift+F1	Toggles the Help pointer.

Creating a Custom Dialog

This section describes the types of controls that the Dialog Editor supports. It also explains how to create controls and initially position them within your dialog, and offers some pointers on creating controls efficiently.

In the next section, Editing a Custom Dialog, you'll learn how to make various types of changes to the controls that you've created—moving and resizing them, assigning labels and accelerator keys, and so forth.

Types of Controls



The Dialog Editor supports the following types of standard Windows controls:

Push button

A command button. The OK, Cancel, and Help buttons are special types of push buttons.

Option button

One of a group of two or more linked buttons that let users select only one from a group of mutually exclusive choices. A group of option buttons works the same way as the buttons on a car radio: because the buttons operate together as a group, clicking an unselected button in the group selects that button and automatically deselects the previously selected button in that group.

Checkbox

A box that users can check or clear to indicate their preference regarding the alternative specified on the checkbox label.

Group box

A rectangular design element used to enclose a group of related controls. You can use the optional group box label to display a title for the controls in the box.

Text

A field containing text that you want to display for the users' information. The text in this field wraps, and the field can contain a maximum of 255 characters. Text controls can either display stand-alone text or be used as labels for text boxes, listboxes, combo boxes, drop-down listboxes, pictures, and picture buttons. You can choose the font in which the text appears.

Text box

A field into which users can enter text (potentially, as much as 32K). By default, this field holds a single line of nonwrapping text. If you choose the Multiline setting in the Text Box Information dialog, this field will hold multiple lines of wrapping text.

Listbox

A displayed, scrollable list from which users can select one item. The currently selected item is highlighted on the list.

Combo box

A text field with a displayed, scrollable list beneath it. Users can either select an item from the list or enter the name of the desired item in the text field. The currently selected item is displayed in the text field. If the item was selected from the scrolling list, it is highlighted there as well.

Drop-down listbox

A field that displays the currently selected item, followed by a downward-pointing arrow, which users can click to temporarily display a scrolling list of items. Once they select an item from the list, the list disappears and the newly selected item is displayed in the field.

Picture

A field used to display a Windows bitmap or metafile.

Picture button

A special type of push, or command, button on which a Windows bitmap or metafile appears.

Note Group boxes, text controls, and pictures are passive elements in a dialog, inasmuch as they are used purely for decorative or informative purposes. Users cannot act upon these controls, and when they tab through the dialog, the focus skips over these controls. You can obtain a Windows bitmap or metafile from a file or from a specified library.

Adding Controls to a Dialog

This section explains how to create controls and determine approximately where they first appear within your dialog. The next section explains how to determine the positioning of controls more precisely. Follow these steps:

1. From the toolbar, choose the button corresponding to the type of control you want to add.

   When you pass the mouse pointer over an area of the display where a control can be placed, the pointer becomes an image of the selected control with crosshairs (for positioning purposes) to its upper left. The name and position of the selected control appear on the status bar. When you pass the pointer over an area of the display where a control cannot be placed, the pointer changes into a circle with a slash through it (the "prohibited" symbol).

Note You can only insert a control within the borders of the dialog you are creating. You cannot insert a control on the dialog's title bar or outside its borders.

2. Place the pointer where you want the control to be positioned and click the mouse button.

   The control you just created appears at the specified location. (To be more specific, the upper left corner of the control will correspond to the position of the pointer's crosshairs at the moment you clicked the mouse button.) The control is surrounded by a thick frame, which means that it is selected, and it may also have a default label.

   After the new control has appeared, the mouse pointer becomes an arrow, to indicate that the toolbar Pick button is active and you can once again select any of the controls in your dialog.

3. To add another control of the same type as the one you just added, press Ctrl+D.

   A duplicate copy of the control appears.

4. To add a different type of control, repeat steps 1 and 2.
5. To reactivate the toolbar Pick button, click the toolbar arrow-shaped button.Or, place the mouse pointer on the title bar of the dialog or outside the borders of the dialog (that is, on any area where the mouse pointer turns into the "prohibited" symbol) and click the mouse button.

   As you plan your dialog, keep in mind that a single dialog can contain no more than 255 controls and that a dialog will not operate properly unless it contains either an OK button, a Cancel button, a push button, or a picture button. (When you create a new custom dialog, an OK button and a Cancel button are provided for you by default.)

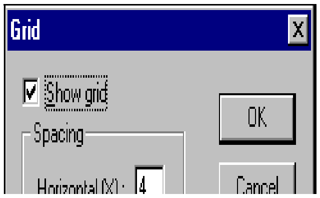
Using the Grid to Help You Position Controls within a Dialog

The preceding subsection explained how to determine approximately where a newly created control will materialize in your dialog. Here, you'll learn how to use the Dialog Editor's grid to help you fine-tune the initial placement of controls.

The area of your dialog in which controls can be placed (that is, the portion of the dialog below the title bar) can be thought of as a grid, with the X (horizontal) axis and the Y (vertical) axis intersecting in the upper left corner (the 0, 0 coordinates). The position of controls can be expressed in terms of X units with respect to the left border of this area and in terms of Y units with respect to the top border. (In fact, the position of controls is expressed in this manner within the dialog template that you produce by working with the Dialog Editor.)

Follow these steps:

1. Press Ctrl+G. The following dialog appears:



2. To see the grid in your dialog, select the Show Grid checkbox.
3. To change the current X and Y settings, enter new values in the X and Y fields.

Note The values of X and Y in the Grid dialog determine the grid's spacing. Assigning smaller X and Y values produces a more closely spaced grid, which enables you to move the mouse pointer in smaller horizontal and vertical increments as you position controls. Assigning larger X and Y values produces the opposite effect on both the grid's spacing and the movement of the mouse pointer. The X and Y settings entered in the Grid dialog remain in effect regardless of whether you choose to display the grid.

4. Click OK or press Enter.

   The Dialog Editor displays the grid with the settings you specified. With the grid displayed, you can line up the crosshairs on the mouse pointer with the dots on the grid to position controls precisely and align them with respect to other controls.

   As you move the mouse pointer over the dialog after you have chosen a control button from the toolbar, the status bar displays the name of the type of control you have selected and continually updates the position of the mouse pointer in X and Y units. (This information disappears if you move the mouse pointer over an area of the screen where a control cannot be placed.) After you click the mouse button to add a control, that control remains selected, and the status bar displays the control's width and height in dialog units as well as its name and position.

Note Dialog units represent increments of the font in which the Dialog Editor creates dialogs (namely, 8 point Helvetica). Each X unit represents an increment equal to 1/4 of that font, and each Y unit represents an increment equal to 1/8 of that font.

Creating Controls Efficiently

Creating dialog controls in random order might seem like the fastest approach. However, the order in which you create controls has some important implications, so a little advance planning can save you a lot of work in the long run.

Here are several points about creating controls that you should keep in mind:

Tabbing order

Users can select dialog controls by tabbing from one control to the next. The order in which you create the controls is what determines the tabbing order. The closer you can come to creating controls in the order in which you want them to receive the tabbing focus, the fewer tabbing-order adjustments you'll have to make later on.

Option button grouping

If you want a series of option buttons to work together as a mutually exclusive group, you must create all the buttons in that group one right after the other, in an unbroken sequence. If you get sidetracked and create a different type of control before you have finished creating all the option buttons in your group, you'll split the buttons into two (or more) separate groups.

Accelerator keys

You can provide easy access to a text box, listbox, combo box, or drop-down listbox by assigning an accelerator key to an associated text control, and you can provide easy access to the controls in a group box by assigning an accelerator key to the group box label. To do this, you must create the text control or group box first, followed immediately by the controls that you want to associate with it. If the controls are not created in the correct order, they will not be associated in your dialog template, and any accelerator key you assign to the text control or group box label will not work properly.

If you don't create controls in the most efficient order, the resulting problems with tabbing order, option button grouping, and accelerator keys usually won't become apparent until you test your dialog. Although you can still fix these problems at that point, it will definitely be more cumbersome. In short, it's easier to prevent (or at least minimize) problems of this sort than to fix them after the fact.

Editing a Custom Dialog

In the preceding section, you learned how to create controls and determine where they initially appear within your dialog. In this section, you'll learn how to make changes to both the dialog and the controls in it. The following topics are included:

- Selecting items so that you can work with them
- Using the Information dialog to check and/or change various attributes of items
- Changing the position and size of items
- Changing titles and labels
- Assigning accelerator keys
- Specifying pictures
- Creating or modifying picture libraries under Windows
- Duplicating and deleting controls
- Undoing editing operations

Selecting Items

In order to edit a dialog or a control, you must first select it. When you select an item, it becomes surrounded by a thick frame, as you saw in the preceding section.

To select a control:

- With the toolbar Pick button active, place the mouse pointer on the desired control and click the mouse button.

Or

- With the Toolbar Pick button active, press the Tab key repeatedly until the focus moves to the desired control.

The control is now surrounded by a thick frame to indicate that it is selected and you can edit it.

To select the dialog:

- With the Toolbar Pick button active, place the mouse pointer on the title bar of the dialog or on an empty area within the borders of the dialog (that is, on an area where there are no controls) and click the mouse button.

Or

- With the Toolbar Pick button active, press the Tab key repeatedly until the focus moves to the dialog.

The dialog is now surrounded by a thick frame to indicate that it is selected and you can edit it.

Using the Information Dialog

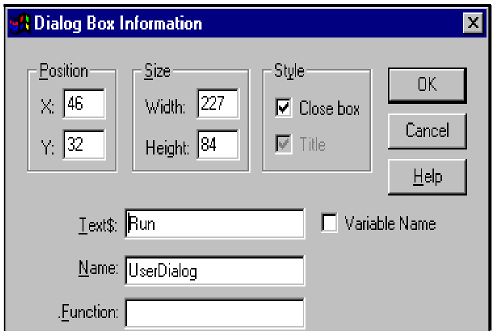
The Information dialog enables you to check and adjust various attributes of controls and dialogs. This subsection explains how to display the Information dialog and provides an overview of the attributes with which it lets you work. In the following subsections, you'll learn more about how to use the Information dialog to make changes to your dialog and its controls.

To see the Information dialog for a dialog:

- With the Toolbar Pick button active, place the mouse pointer on an area of the dialog where there are no controls and double-click the mouse button.

Or

- With the Toolbar Pick button active, select the dialog and either click the toolbar Information button, press Enter, or press Ctrl+I. The following dialog appears:



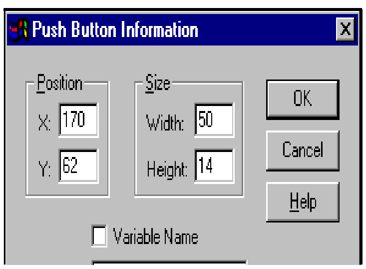
To display the Information dialog for a control:

- With the Toolbar Pick button active, place the mouse pointer on the desired control and double-click the mouse button.

Or

- With the Toolbar Pick button active, select the control and either click the toolbar Information button, press Enter, or press Ctrl+I.

The Information dialog corresponding to the control you selected appears:



The following lists show the attributes that you can change with the Dialog Information and Information dialogs for the various controls. In some cases (specified below), it's mandatory to fill in the fields in which the attributes are specified—that is, you must either leave the default information in these fields or replace it with more meaningful information, but you can't leave the fields empty. In other cases, filling in these fields is optional.

Note A quick way to determine whether it's mandatory to fill in a particular Information dialog field is to see whether the OK button becomes grayed out when you delete the information in that field. If it does, then you *must* fill in that field.

In many cases, you could simply leave the generic-sounding default information in the Information dialog fields and worry about replacing it with more meaningful information after you paste the dialog template into your macro. However, if you take a few moments to replace the default information with something specific when you first create your dialog, not only will you save yourself some work later on but you may also find that your changes make the code produced by the Dialog Editor more readily comprehensible and thus easier to work with.

Dialog Attributes

Mandatory/ Optional	Attribute
Optional	**Position:** X and Y coordinates on the display, in dialog units
Mandatory	**Size:** width and height of the dialog, in dialog units
Optional	**Style:** options that allow you to determine whether the close box and title bar are displayed
Optional	**Text$:** text displayed on the title bar of the dialog
Mandatory	**Name:** name by which you refer to this dialog template in your code
Optional	.**Function:** name of a function in your dialog
Optional	**Picture Library:** picture library from which one or more pictures in the dialog are obtained

Control Attributes

Mandatory/ Optional	Control(s) Affected	Attribute
Mandatory	All controls	**Position:** X and Y coordinates within the dialog, in dialog units
Mandatory	All controls	**Size:** width and height of the control, in dialog units
Optional	Push button, option button, checkbox, group box, and text	**Text$:** text displayed on a control
Optional	Help button	**FileName$:** name of the help file invoked when the user clicks this button
Optional	Text	**Font:** font in which text is displayed
Optional	Text box	**Multiline:** option that allows you to determine whether users can enter a single line of text or multiple lines
Optional	OK button, Cancel button, push button, option button, group box, and text	.**Identifier:** name by which you refer to a control in your code

Mandatory/ Optional	Control(s) Affected	Attribute
Mandatory	Checkbox, text box, listbox, combo box, drop-down listbox, and help button	.**Identifier:** name by which you refer to a control in your code; also contains the result of the control after the dialog has been processed
Optional	Picture, picture button	.**Identifier:** name of the file containing a picture that you want to display or the name of a picture that you want to display from a specified picture library
Optional	Picture	**Frame:** option that allows you to display a 3-D frame
Mandatory	Listbox, combo box, and drop-down listbox	**Array$:** name of an array variable in your code
Mandatory	Option button	.**Option Group:** name by which you refer to a group of option buttons in your code

Position and Size

This section explains how the Dialog Editor helps you keep track of the location and dimensions of dialogs and controls, and presents several ways to move and resize these items.

Keeping Track of Position and Size

The Dialog Editor's display can be thought of as a grid, in which the X (horizontal) axis and the Y (vertical) axis intersect in the upper left corner of the display (the 0, 0 coordinates). The position of the dialog you are creating can be expressed in terms of X units with respect to the left border of the parent window and in terms of Y units with respect to the top border.

When you select a dialog or control, the status bar displays its position in X and Y units as well as its width and height in dialog units. Each time you move or resize an item, the corresponding information on the status bar is updated. You can use this information to position and size items more precisely.

The Dialog Editor provides several ways to reposition dialogs and controls.

To reposition an item with the mouse:

1. With the Toolbar Pick button active, place the mouse pointer on an empty area of the dialog or on a control.
2. Click the mouse button and drag the dialog or control to the desired location.

Note The increments by which you can move a control with the mouse are governed by the grid setting. For example, if the grid's X setting is 4 and its Y setting is 6, you'll be able to move the control horizontally only in increments of 4 X units and vertically only in increments of 6 Y units. This feature is handy if you're trying to align controls in your dialog. If you want to move controls in smaller or larger increments, press Ctrl+G to display the Grid dialog and adjust the X and Y settings.

To reposition an item with the arrow keys:

1. Select the dialog or control that you want to move.
2. Press an arrow key once to move the item by 1 X or Y unit in the desired direction. Or, click an arrow key to "nudge" the item steadily along in the desired direction.

Note When you reposition an item with the arrow keys, a faint, partial afterimage of the item may remain visible in the item's original position. These afterimages are rare and will disappear once you test your dialog.

To reposition a dialog with the Information dialog:

1. Display the Information dialog.
2. Change the X and Y coordinates in the Position group box. Or, leave the X and/or Y coordinates blank.
3. Click OK or press Enter.

   If you specified X and Y coordinates, the dialog moves to that position. If you left the X coordinate blank, the dialog will be centered horizontally relative to the parent window of the dialog when the dialog is run. If you left the Y coordinate blank, the dialog will be centered vertically relative to the parent window of the dialog when the dialog is run.

To reposition a control with the Information dialog:

1. Display the Information dialog for the control that you want to move.
2. Change the X and Y coordinates in the Position group box.
3. Click OK or press Enter.

   The control moves to the specified position.

Note When you move a dialog or control with the arrow keys or with the Information dialog, the item's movement is not restricted to the increments specified in the grid setting. When you attempt to test a dialog containing hidden controls (i.e., controls positioned entirely outside the current borders of your dialog), the Dialog Editor displays a message advising you that there are controls outside the dialog's borders and asks whether you wish to proceed with the test. If you proceed, the hidden controls will be disabled for testing purposes. (Testing dialogs is discussed later in the chapter.)

Dialogs and controls can be resized either by directly manipulating them with the mouse or by using the Information dialog. Certain controls can also be resized automatically to fit the text displayed on them.

To resize an item with the mouse:

1. With the Toolbar Pick button active, select the dialog or control that you want to resize.
2. Place the mouse pointer over a border or corner of the item.
3. Click the mouse button and drag the border or corner until the item reaches the desired size.

To resize an item with the Information dialog:

1. Display the Information dialog for the dialog or control that you want to resize.
2. Change the Width and Height settings in the Size group box.
3. Click OK or press Enter.

   The dialog or control is resized to the dimensions you specified.

To resize selected controls automatically:

1. With the Toolbar Pick button active, select the option button, text control, push button, checkbox, or text box that you want to resize.
2. Press F2. The borders of the control expand or contract to fit the text displayed on it.

Note Windows metafiles always expand or contract proportionally to fit within the picture control or picture button control containing them. In contrast, Windows bitmaps are of a fixed size. If you place a bitmap in a control that is smaller than the bitmap, the bitmap is clipped off on the right and bottom. If you place a bitmap in a control that is larger than the bitmap, the bitmap is centered within the borders of the control. Picture controls and picture button controls must be resized manually.

Changing Titles and Labels

By default, when you begin creating a dialog, its title reads "Untitled," and when you first create group boxes, option buttons, push buttons, text controls, and checkboxes, they have generic-sounding default labels, such as "Group Box" and "Option Button."

To change a dialog title or a control label:

1. Display the Information dialog for the dialog whose title you want to change or for the control whose label you want to change.
2. Enter the new title or label in the Text$ field.

Note Dialog titles and control labels are optional. Therefore, you can leave the Text$ field blank.

3. If the information in the Text$ field should be interpreted as a variable name rather than a literal string, select the Variable Name checkbox.
4. Click OK or press Enter. The new title or label appears on the title bar or on the control.

   Although OK and Cancel buttons also have labels, you cannot change them. The remaining controls (text boxes, listboxes, combo boxes, drop-down listboxes, pictures, and picture buttons) don't have their own labels, but you can position a text control above or beside these controls to serve as a de facto label for them.

Assigning Accelerator Keys

Accelerator keys enable users to access dialog controls simply by pressing Alt plus a specified letter. Users can employ accelerator keys to choose a push button or an option button; toggle a checkbox on or off; and move the insertion point into a text box or group box or to the currently selected item in a listbox, combo box, or drop-down listbox.

An accelerator key is essentially a single letter that you designate for this purpose from a control's label. You can assign an accelerator key directly to controls that have their own label (option buttons, push buttons, checkboxes, and group boxes). (You can't assign an accelerator key to OK and Cancel buttons because, as noted above, their labels can't be edited.) You can create a de facto accelerator key for certain controls that don't have their own labels (text boxes, listboxes, combo boxes, and drop-down listboxes) by assigning an accelerator key to an associated text control.

To assign an accelerator key:

1. Display the Information dialog for the control to which you want to assign an accelerator key.
2. In the ***`Text$`*** field, type an ampersand (&) before the letter you want to designate as the accelerator key.
3. Click OK or press Enter.

The letter you designated is now underlined on the control's label, and users will be able to access the control by pressing Alt plus the underlined letter.

Note Accelerator key assignments must be unique within a particular dialog. If you attempt to assign the same accelerator key to more than one control, the Dialog Editor displays a reminder that letter has already been assigned.

If, for example, you have a push button whose label reads **`Apply`**, you can designate **`A`** as the accelerator key by displaying the Push Button Information dialog and typing **`&Apply`** in the ***`Text$`*** field. When you press Enter, the button label says Apply, and users will be able to choose the button by pressing Alt+A.

Note In order for such a default accelerator key to work properly, the text control or group box label to which you assign the accelerator key must be associated with the control(s) to which you want to provide user access. That is, in the dialog template, the description of the text control or group box must immediately precede the description of the control(s) that you want associated with it. The simplest way to establish such an association is to create the text control or group box first, followed immediately by the associated control(s).

Specifying Pictures

In the preceding section, you learned how to add picture controls and picture button controls to your dialog. But these controls are nothing more than empty outlines until you specify the pictures that you want them to display.

A picture control or picture button control can display a Windows bitmap or metafile, which you can obtain from a file or from a specified library. (Refer to the following subsection for information on creating or modifying picture libraries under Windows.)

To specify a picture from a file:

1. Display the Information dialog for the picture control or picture button control whose picture you want to specify.
2. In the Picture source option button group, select File.
3. In the Name$ field, enter the name of the file containing the picture you want to display in the picture control or picture button control.

Note Click Browse to see the Select a Picture File dialog and use it to find the file.

4. Click OK or press Enter. The picture control or picture button control now displays the picture you specified.

To specify a picture from a picture library:

1. Display the Information dialog.
2. In the Picture Library field, specify the name of the picture library that contains the picture(s) you want to display in your dialog.

Note Click Browse to see the Select a Picture Library dialog and use it to find the file. If you specify a picture library in the Information dialog, all the pictures in your dialog must come from this library.

3. Click OK or press Enter.
4. Display the Information dialog for the picture control or picture button control whose picture you want to specify.
5. In the Picture source option button group, select Library.
6. In the Name$ field, enter the name of the picture you want to display on the picture control or picture button control. (This picture must be from the library that you specified in step 2.)
7. Click OK button or Enter. The picture control or picture button control now displays the picture you specified.

Creating or Modifying Picture Libraries under Windows

The **`Picture`** statement allows images to be specified as individual picture files or as members of a picture library, which is a DLL that contains a collection of pictures. Both Windows bitmaps and metafiles are supported. You can obtain a picture library either by creating a new one or by modifying an existing one, as described below.

Each image is placed into the DLL as a resource identified by its unique resource identifier. This identifier is the name used in the **`Picture`** statement to specify the image.

The following resource types are supported in picture libraries:

Resource Type	Description
2	Bitmap. This is defined in windows.h as **RT_BITMAP.**
256	Metafile. Since there is no resource type for metafiles, 256 is used.

To create a picture library under Windows:

1. Create a C file containing the minimal code required to establish a DLL. The following code can be used:

```
#include <windows.h>
int CALLBACK LibMain(
   HINSTANCE hInstance,
   WORD wDataSeg,
   WORD wHeapSz,
   LPSTR lpCmdLine) {
   UnlockData(0);
   return 1;
}
```

2. Use the following code to create a DEF file for your picture library:

```
LIBRARY
DESCRIPTION "My Picture Library"
EXETYPE WINDOWS
CODE LOADONCALL MOVABLE DISCARDABLE
DATA PRELOAD MOVABLE SINGLE
HEAPSIZE 1024
```

3. Create a resource file containing your images. The following example shows a resource file using a bitmap called sample.bmp and a metafile called usa.wmf.

```
#define METAFILE 256
USA METAFILE "usa.wmf"
MySample BITMAP "sample.bmp"
```

4. Create a make file that compiles your C module, creates the resource file, and links everything together.

To modify an existing picture library:

1. Make a copy of the picture library you want to modify.
2. Modify the copy by adding images using a resource editor such as Borland's Resource Workshop or Microsoft's App Studio.

Note When you use a resource editor, you need to create a new resource type for metafiles (with the value 256).

Duplicating Controls

1. Select the control that you want to duplicate.
2. Press Ctrl+D. A duplicate copy of the selected control appears in your dialog.

3. Repeat step 2 as many times as necessary to create the desired number of duplicate controls.

   Duplicating is a particularly efficient approach if you need to create a group of controls, such as a series of option buttons or checkboxes. Simply create the first control in the group and then, while the newly created control remains selected, repeatedly press Ctrl+D until you have created the necessary number of copies.

The Dialog Editor also enables you to delete single controls or even clear the entire dialog.

Deleting Controls

To delete a single control:

1. Select the control you want to delete.
2. Press Del.

   The selected control is removed from your dialog.

To delete all the controls in a dialog:

1. Select the dialog.
2. Press Del.
3. If the dialog contains more than one control, the Dialog Editor prompts you to confirm that you want to delete all controls. Click the Yes button or press Enter.

   All the controls disappear, but the dialog's title bar and close box (if displayed) remain unchanged.

Undoing Editing Operations

You can undo editing operations that produce a change in your dialog, including:

- The addition of a control
- The insertion of one or more controls from the Clipboard
- The deletion of a control
- Changes made to a control or dialog, either with the mouse or with the Information dialog

You cannot undo operations that don't produce any change in your dialog, such as selecting controls or dialogs and copying material to the Clipboard.

To undo an editing operation:

- Press Ctrl+Z.

Your dialog is restored to the way it was before you performed the editing operation.

Editing an Existing Dialog

There are three ways to edit an existing dialog:

- You can copy the template of the dialog you want to edit from a macro to the Clipboard and paste it into the Dialog Editor.
- You can use the capture feature to "grab" an existing dialog from another application and insert a copy of it into the Dialog Editor.
- You can open a dialog template file that has been saved on a disk. Once you have the dialog displayed in the Dialog Editor's application window, you can edit it using the methods described earlier in the chapter.

Pasting an Existing Dialog into the Dialog Editor

You can use the Dialog Editor to modify the macro statements that correspond to an entire dialog or to one or more dialog controls.

If you want to modify a dialog template contained in your macro, here's how to select the template and paste it into the Dialog Editor for editing.

To paste an existing dialog into the Dialog Editor:

1. Copy the entire dialog template (from the **`Begin Dialog`** instruction to the **`End Dialog`** instruction) from your macro to the Clipboard.
2. Open the Dialog Editor.
3. Press Ctrl+V.
4. When the Dialog Editor asks whether you want to replace the existing dialog, click the Yes button.

   The Dialog Editor creates a new dialog corresponding to the template contained on the Clipboard.

If you want to modify the macro statements that correspond to one or more dialog controls, here's how to select the statements and paste them into the Dialog Editor for editing.

To paste one or more controls from an existing dialog into the Dialog Editor:

1. Copy the description of the control(s) from your macro to the Clipboard.
2. Open the Dialog Editor.
3. Press Ctrl+V.

   The Dialog Editor adds to your current dialog one or more controls corresponding to the description contained on the Clipboard.

Note When you paste a dialog template into the Dialog Editor, the tabbing order of the controls is determined by the order in which the controls are described in the template. When you paste one or more controls into the Dialog Editor, they will come last in the tabbing order, following the controls that are already present in the current dialog.

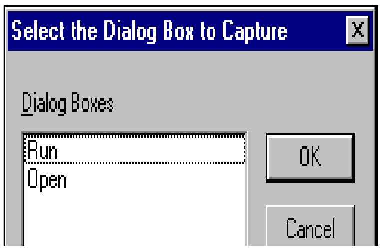
If there are any errors in the statements that describe the dialog or controls, the Dialog Translation Errors dialog will appear when you attempt to paste these statements into the Dialog Editor. This dialog shows the lines of code containing the errors and provides a brief description of the nature of each error.

Capturing a Dialog

Here's how to capture the standard Windows controls from any standard Windows dialog in another application and insert those controls into the Dialog Editor for editing.

To capture an existing standard Windows dialog:

1. Display the dialog you want to capture.
2. Open the Dialog Editor.
3. Select File>Capture Dialog. The Dialog Editor displays a dialog that lists all open dialogs that it is able to capture:



4. Select the dialog you want to capture, then click OK. The Dialog Editor now displays the standard Windows controls from the target dialog.

Note The Dialog Editor only supports standard Windows controls and standard Windows dialogs. Therefore, if the target dialog contains both standard Windows controls and custom controls, only the standard Windows controls will appear in the Dialog Editor's application window. If the target dialog is not a standard Windows dialog, you will be unable to capture the dialog or any of its controls.

Opening a Dialog Template File

Here's how to open any dialog template file that has been saved on a disk so you can edit the template in the Dialog Editor.

To open a dialog template file:

1. Select File>Open. The Open Dialog File dialog appears.
2. Select the file containing the dialog template that you want to edit and click the OK button.

   The Dialog Editor creates a dialog from the statements in the template and displays it in the application window.

Note If there are any errors in the statements that describe the dialog, the Dialog Translation Errors dialog will appear when you attempt to load the file into the Dialog Editor. This dialog shows the lines of code containing the errors and provides a brief description of the nature of each error.

Testing a Dialog

The Dialog Editor lets you run your edited dialog for testing purposes. When you click the toolbar Test Dialog button, your dialog comes alive, which gives you an opportunity to make sure it functions properly and fix any problems before you incorporate the dialog template into your macro.

Before you run your dialog, take a moment to look it over for basic problems such as the following:

- Does the dialog contain a command button—that is, a default OK or Cancel button, a push button, or a picture button?
- Does the dialog contain all the necessary push buttons?
- Does the dialog contain a Help button if one is needed?
- Are the controls aligned and sized properly?
- If there is a text control, is its font set properly?
- Are the close box and title bar displayed (or hidden) as you intended?
- Are the control labels and dialog title spelled and capitalized correctly?
- Do all the controls fit within the borders of the dialog?
- Could you improve the design of the dialog by adding one or more group boxes to set off groups of related controls?
- Could you clarify the purpose of any unlabeled control (such as a text box, listbox, combo box, drop-down listbox, picture, or picture button) by adding a text control to serve as a de facto label for it?
- Have you made all the necessary accelerator key assignments?

After you've fixed any elementary problems, you're ready to run your dialog so you can check for problems that don't become apparent until a dialog is activated.

Testing your dialog is an iterative process that involves running the dialog to see how well it works, identifying problems, stopping the test and fixing those problems, then running the dialog again to make sure the problems are fixed and to identify any additional problems, and so forth—until the dialog functions the way you intend. Here's how to test your dialog and fine-tune its performance.

To test your dialog:

1. Click the toolbar Test Dialog button or press F5. The dialog becomes operational, and you can check how it functions.
2. To stop the test, click the toolbar Test Dialog button, press F5, or double-click the dialog's close box (if it has one).
3. Make any necessary adjustments to the dialog.
4. Repeat steps 1–3 as many times as you need in order to get the dialog working properly.

When testing a dialog, you can check for operational problems such as the following:

Tabbing order

When you press the Tab key, does the focus move through the controls in a logical order? (Remember, the focus skips over items that users cannot act upon, including group boxes, text controls, and pictures.)

When you paste controls into your dialog, the Dialog Editor places their descriptions at the end of your dialog template, in the order in which you paste them in. Therefore, you can use a simple cut-and-paste technique to adjust the tabbing order. First, click the toolbar Test Dialog button to end the test and then, proceeding in the order in which you want the controls to receive the focus, select each control, cut it from the dialog (by pressing Ctrl+X), and immediately paste it back in again (by pressing Ctrl+V). The controls will now appear in the desired order in your template and will receive the tabbing focus in that order.

Option button grouping

Are the option buttons grouped correctly? Does selecting an unselected button in a group automatically deselect the previously selected button in that group?

To merge two groups of option buttons into a single group, click the toolbar Test Dialog button to end the test and then use the Option Button Information dialog to assign the same .Option Group name for all the buttons that you want included in that group.

Text box functioning

Can you enter only a single line of nonwrapping text, or can you enter multiple lines of wrapping text?

If the text box doesn't behave the way you intended, click the toolbar Test Dialog button to end the test; then display the Text Box Information dialog and select or clear the Multiline checkbox.

Accelerator keys

If you have assigned an accelerator key to a text control or group box in order to provide user access to a text box, listbox, combo box, drop-down listbox, or group box, do the accelerator keys work properly? That is, if you press Alt + the designated accelerator key, does the insertion point move into the text box or group box or to the currently selected item in the listbox, combo box, or drop-down listbox?

If the accelerator key doesn't work properly, it means that the text box, listbox, combo box, drop-down listbox, or group box is not associated with the text control or group box to which you assigned the accelerator key—that is, in your dialog template, the description of the text control or group box does not immediately precede the description of the control(s) that should be associated with it. As with tabbing-order problems (discussed above), you can fix this problem by using a simple cut-and-paste technique to adjust the order of the control descriptions in your template. First, click the toolbar Test Dialog button to end the test; then cut the text control or group box from the dialog and immediately paste it back in again; and finally, do the same with each of the controls that should be associated with the text control or group box. The controls will now appear in the desired order in your template, and the accelerator keys will work properly.

Incorporating a Dialog into a Macro

Once you have created a dialog or dialog controls, you can paste it into your macro via the Clipboard. Follow these steps.

To incorporate a dialog or control into your macro:

1. Select the dialog or control that you want to incorporate into your macro.
2. Press Ctrl+C.
3. Open your macro and paste in the contents of the Clipboard at the desired point.

   You can also select File>Save As on the Dialog Editor and save the dialog to a .DLG file. Later you can open the macro in the Macro Editor and the saved dialog in the Dialog Editor, and copy the dialog into the macro.

   The dialog template or control is now described in statements in your macro.

Using Dialogs

After using the Dialog Editor to insert a custom dialog template into your macro, you'll need to make the following modifications to your macro:

1. Create a dialog record with the **`Dim`** statement.
2. Put information into the dialog by assigning values to its controls.
3. Display the dialog with either the **`Dialog()`** function or the **`Dialog`** statement.
4. Retrieve values from the dialog after the user closes it.

Creating a Dialog Record

To store the values retrieved from a custom dialog, create a dialog record with a **Dim** statement using the following syntax:

```
Dim DialogRecord As DialogVariable
```

Here are some examples of how to create dialog records:

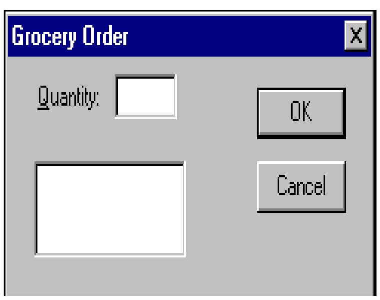
```
Dim b As UserDialog    'Define a dialog record "b"
Dim PlayCD As CDDialog 'Define dialog record PlayCD.
```

Here is a sample macro that illustrates how to create a dialog record named b within a dialog template named **UserDialog**. Notice that the order of the statements within the macro is: the dialog template precedes the statement that creates the dialog record, and the **Dialog** statement follows both of them.

```
Sub Main
'!
    Dim ListBox1$()    'Initialize listbox array.
    'Define the dialog template.
    Begin Dialog UserDialog ,,163,94,"Grocery Order"
        Text 13,6,32,8,"&Quantity:",.Text1
        TextBox 48,4,28,12,.TextBox1
        ListBox 12,28,68,32,ListBox1$,.ListBox1
        OKButton 112,8,40,14
        CancelButton 112,28,40,14
    End Dialog
    Dim b As UserDialog    'Create the dialog record.
    Dialog b    'Display the dialog.
End Sub
```

Putting Information into the Dialog

When you open and run the sample macro shown in the preceding subsection, you see a dialog like the following:



To put information into this dialog, assign values to its controls by modifying the statements in your macro that are responsible for displaying those controls to the user. The following table lists the dialog controls to which you can assign values and the types of information you can control:

Control(s)	Types of Information
Listbox, drop-down listbox, combo box	Items
Text box	Default text
Checkbox	Values

The following sections explain how to define and fill an array, set the default text in a text box, and set the initial focus and tab order for the controls in a custom dialog.

Defining and Filling an Array

You can store items in the listbox shown in the example above by creating an array and then assigning values to the elements of the array. For example, you could include the following lines to initialize an array with three elements and assign the names of three common fruits to these elements of your array:

```
Dim ListBox1$(3)    'Initialize listbox array.
ListBox1$(0) = "Apples"
ListBox1$(1) = "Oranges"
ListBox1$(2) = "Pears"
```

Setting Default Text in a Text Box

You can set the default value of the text box in your macro to 12 with the following assignment statement. This assignment must follow the definition of the dialog record but precede the statement or function that displays the custom dialog.

```
b.TextBox1 = "12"
```

Setting the Initial Focus and Controlling the Tabbing Order

You can determine which control has the focus when your custom dialog appears as well as the tabbing order between controls by understanding two rules. First, the focus in a custom dialog is always set initially to the first control to appear in the dialog template. Second, the order in which subsequent controls appear within the dialog template determines the tabbing order. That is, pressing the Tab key will change the focus from the first control to the second one, pressing the Tab key again will change the focus to the third control, and so on.

Displaying the Custom Dialog

To display a custom dialog, use either the `Dialog()` function or the `Dialog` statement.

Using the Dialog() Function

Use the `Dialog()` function to determine how the user closed your custom dialog. For example, the following statement returns a value when the user clicks an OK button or a Cancel button or takes another action:

```
response% = Dialog(b)
```

The **`Dialog()`** function returns any of the following values:

Value Returned	If
–1	The user clicked the OK button.
0	The user clicked the Cancel button.
>0	The user clicked a push button. The returned number represents which button was clicked based on its order in the dialog template (1 is the first push button, 2 is the second push button, and so on).

Using the Dialog Statement

Use the **`Dialog`** statement when you don't need to determine how the user closed your dialog. You can still retrieve other information from the dialog record, such as the value of a listbox or other dialog control. The following is an example of the correct use of the **`Dialog`** statement:

```
Dialog b
```

Retrieving Values from the Custom Dialog

After displaying a custom dialog, the macro must retrieve the values of the dialog controls by referencing the appropriate identifiers in the dialog record. The following example uses several of the techniques described earlier to explain this process.

In this macro, the array named **`ListBox1`** is filled with three elements ("**`Apples`**", "**`Oranges`**", and "**`Pears`**"). The default value of **`TextBox1`** is set to 12. A variable named **`response`** is used to store information about how the custom dialog was closed. An identifier named **`ListBox1`** is used to determine whether the user chose "**`Apples`**", "**`Oranges`**", or "**`Pears`**" in the listbox named **`ListBox$`**. Finally, a **`Select Case...End Select`** statement is used to display a message box appropriate to the manner in which the user dismissed the dialog.

```
Sub Main
'!
    Dim ListBox1$(2)    'Initialize listbox array.
    Dim response%
    ListBox1$(0) = "Apples"
    ListBox1$(1) = "Oranges"
    ListBox1$(2) = "Pears"
    Begin Dialog UserDialog ,,163,94,"Grocery Order"
        'First control gets focus.
        Text 13,6,32,8,"&Quantity:",.Text1
        TextBox 48,4,28,12,.TextBox1
        ListBox 12,28,68,32,ListBox1$,.ListBox1
        OKButton 112,8,40,14
        CancelButton 112,28,40,14
    End Dialog
    Dim b As UserDialog    'Create the dialog record.
    'Set default value of the text box to 1 dozen.
    b.TextBox1 = "12"
    response% = Dialog(b)    'Display the dialog.
```

```
    Select Case response%
        Case -1
            Fruit$ = ListBox1$(b.ListBox1)
            MsgBox "Thank you for ordering " + _
            b.TextBox1 + " " + Fruit$ + "."
        Case 0
            MsgBox "Your order has been canceled."
    End Select
End Sub
```

Using a Dynamic Dialog in a Macro

The preceding section explained how to use a custom dialog in your macro. As you learned, you can retrieve the values from dialog controls after the user dismisses the dialog by referencing the identifiers in the dialog record.

You can also retrieve values from a custom dialog while the dialog is displayed, using a feature of called dynamic dialogs.

The following macro illustrates the most important concepts you'll need to understand in order to create a dynamic dialog in your macro:

```
'Dim "Fruits" and "Vegetables" arrays here to make them
'accessible to all procedures.
Dim Fruits(2) As String
Dim Vegetables(2) As String
'Dialog procedure--must precede the procedure that defines
'the custom dialog.
Function DialogControl(ctrl$, action%, suppvalue%) As Integer
    Select Case action%
        Case 1
            'Fill listbox with items before dialog is visible.
            DlgListBoxArray "ListBox1", fruits
            'Set default value to first item in listbox.
            DlgValue "ListBox1", 0
        Case 2
            'Fill the listbox with names of fruits or vegetables
            'when the user selects an option button.
            If ctrl$ = "OptionButton1" Then
                DlgListBoxArray "ListBox1", fruits
                DlgValue "ListBox1", 0
            ElseIf ctrl$ = "OptionButton2" Then
                DlgListBoxArray "ListBox1", vegetables
                DlgValue "ListBox1", 0
            End If
        End Select
End Function
Sub Main
'!
    'Initialize array for use by ListBox statement in template.
    Dim ListBox1$()
    Dim Produce$
    'Assign values to elements in the Fruits and Vegetables arrays.
    Fruits(0) = "Apples"
    Fruits(1) = "Oranges"
    Fruits(2) = "Pears"
```

```
    Vegetables(0) = "Carrots"
    Vegetables(1) = "Peas"
    Vegetables(2) = "Lettuce"
    'Define the dialog template.
    Begin Dialog UserDialog ,,163,94,"Grocery Order", .DialogControl
        Text 13,6,32,8,"&Quantity:",.Text1'First control
            'in template gets the focus.
        TextBox 48,4,28,12,.TextBox1
    ListBox 12,28,68,32,ListBox1$,.ListBox1
    OptionGroup .OptionGroup1
        OptionButton 12,68,48,8,"&Fruit",.OptionButton1
        OptionButton 12,80,48,8,"&Vegetables",.OptionButton2
        OKButton 112,8,40,14
        CancelButton 112,28,40,14
    End Dialog
    Dim b As UserDialog    'Create the dialog record.
    'Set the default value of the text box to 1 dozen.
    b.TextBox1 = "12"
    response% = Dialog(b)    'Display the dialog.
    Select Case response%
        Case -1
            If b.OptionGroup1 = 0 Then
                produce$ = fruits(b.ListBox1)
            Else
                produce$ = vegetables(b.ListBox1)
            End If
            MsgBox "Thank you for ordering " & _
                b.TextBox1 & " " & produce$ & "."
        Case 0
            MsgBox "Your order has been canceled."
    End Select
End Sub
```

The remainder of this section explains how to make a dialog dynamic by examining the workings of this sample macro.

Making a Dialog Dynamic

The first thing to notice about the preceding macro, which is a more complex variation of the macro described earlier in this chapter, is that an identifier named ***.DialogControl*** has been added to the **Begin Dialog** statement. As you will learn in the following subsection, this parameter to the **Begin Dialog** statement tells the compiler to pass control to a function procedure named **DialogControl**.

Using a Dialog Function

Before the compiler displays a custom dialog by executing a **Dialog** statement or **Dialog()** function, it must first initialize the dialog. During this initialization process, the compiler checks to see whether there is a dialog function defined in the dialog template. If so, it gives control to the dialog function, allowing the macro to carry out certain actions, such as hiding or disabling dialog controls.

After completing its initialization, the compiler displays the custom dialog. When the user selects an item in a listbox, clears a checkbox, or carries out certain other actions within the dialog, the compiler will again call the dialog function.

In fact, the compiler also calls the dialog function repeatedly even while the user is not interacting with the dialog. You can use this fact to update a dialog continuously.

Responding to User Actions

A dialog function can respond to six types of user actions:

Action	Description
1	This action is sent immediately before the dialog is shown for the first time.
2	This action is sent when: • A button is clicked, such as OK, Cancel, or a push button. • A checkbox's state has been modified. • An option button is selected. In this case, ***`ControlName$`*** contains the name of the option button that was clicked, and ***`SuppValue`*** contains the index of the option button within the option button group (0 is the first option button, 1 is the second, and so on). • The current selection is changed in a listbox, drop-down listbox, or combo box. In this case, ***`ControlName$`*** contains the name of the listbox, combo box, or drop-down listbox, and ***`SuppValue`*** contains the index of the new item (0 is the first item, 1 is the second, and so on).
3	This action is sent when the content of a text box or combo box has been changed *and* that control loses focus.
4	This action is sent when a control gains the focus.
5	This action is sent continuously when the dialog is idle.
6	This action is sent when the dialog is moved.

Using objects in an external OLE application

When SmarTerm is operated through an external OLE Automation controller, only those macro commands relating directly to the SmarTerm objects are available. This means that another application can use commands such as **`Session.Circuit.Connect`**, but not commands such as **`LTrim$`** or **`Open`**. This is not a great hardship, however, since programming commands not directly related to the operation of SmarTerm should be available in the macro language for the controlling application.

To provide another application with OLE access to SmarTerm objects, you must include some basic definitions in the controlling application's code. The following *preamble* will provide a controlling application complete access to the SmarTerm objects:

```
' acquire access to SmarTerm for automation control
    Dim Application as Object
    Set Application = CreateObject("SmarTerm.Application")
```

```
' initialize a Session object by opening a session file
    Dim Session as Object
    Set Session = Application.Sessions.Open("Session1.STW")

' initialize a Circuit object for access to communications
' features
    Dim Circuit as Object
    Set Circuit = Session.Circuit

' initialize a Transfer object for access to file transfer
' features
    Dim Transfer as Object
    Set Transfer = Session.Transfer
```

Once you have included this preamble, you can then construct the rest of the controlling application's macro code to access SmarTerm objects exactly as described in the online help.

Communicating with a host

Since the primary purpose of terminal emulation software is to communicate with a host, a high proportion of the macro commands support host communication tasks, such as connecting to the host, transferring data, and handling user interaction with the host. These tasks are handled by three SmarTerm objects: **`Circuit`**, **`Session`**, and **`Transfer`**. In this section we discuss common host communication tasks and provide generalized sample macros that should help you design your own macros specific to the tasks you need to accomplish.

Handling host connections

The macro commands that control host connection are all properties or methods of the SmarTerm **`Circuit`** object. These commands fall into two groups:

- Connection commands (such as **`Circuit.Connect`**, **`Circuit.Connected`**, and **`Circuit.Disconnect`**), which are common to all communication methods
- Setup commands, which are unique to each communication method

For the most part, it is probably easiest to set up SmarTerm session files with the appropriate connection information rather than use macro commands to do it. For one thing, it's easier to run SmarTerm and then save the session file than it is to debug a macro that sets up a complicated session type. Also, between the Session_Connect built-in macro (see “Session_Connect macro” on page 33) and the fact that you can have the session connect automatically when you open it, you may not need a special macro to handle your connection at all.

However, not everyone's needs are so easily satisfied. For example, suppose that you need to connect to multiple telnet hosts that all use the same display and keyboard settings, but you can only make one connection at a time due to network cost constraints. One way in which you can do this is to set up a single session file with the common display and keyboard settings, then provide that session file with SmarTerm buttons that allow you to connect to several hosts. Follow these steps:

1. Create a session. When asked for the connection settings, pick one of the hosts you routinely connect to.
2. Set up the display, terminal type, keyboard map, and so forth, the way you want them. Then save the session file.
3. Now use Tools>SmarTerm buttons to create a set of buttons, one for each host. Attach to each button a macro like the following:

```
Sub Connect_ThisHost
'! Use this macro to connect to ThisHost.com

If Circuit.Connected = True Then 'Are we connected?
    If Circuit.TelnetHostname = "ThisHost.com" Then
        End 'Already connected to target host--quit!using se
    Else
        Session.Send "Logout" 'log off other host
        Circuit.Disconnect
    End If
End If
    Circuit.Telnet.Hostname = "ThisHost.com"
    Circuit.Connect
End Sub
```

   For each SmarTerm button, substitute the name of the new host for the sample text "ThisHost" and "ThisHost.com". You may also need to change the logout command.

4. When you have created all your buttons, save them and save the session. From now on, when you open the session you will have a set of SmarTerm buttons that allow you to switch from host to host.

Possible improvements

There are several improvements you could make to the host connection macro. First, you can add error-checking to handle situations in which things do not go as planned. This is simplified by the fact that the **`Circuit`** methods **`Circuit.Connect`** and **`Circuit.Disconnect`** are functions that return either **`True`** or **`False`**, depending on whether they succeed or not. If we add a check for success into the sample above, we get the following macro.

```
Sub Connect_ThisHost
'! Use this macro to connect to ThisHost.com
' Improved to check for success on connect and disconnect

If Circuit.Connected = True Then 'Are we connected?
    If Circuit.TelnetHostname = "ThisHost.com" Then
        End 'Already connected to target host--quit!
    Else
        Session.Send "Logout" 'log off other host
        'Unable to disconnect?
If Circuit.Disconnect = False Then
            Session.Echo "Unable to disconnect from " +_
                Circuit.Telnethostname + ". Please contact IS."
```

```
            End 'Quit!
        End If
    End If
End If
    Circuit.Telnet.Hostname = "ThisHost.com"
    If Circuit.Connect = False Then ' Unable to connect?
        Session.Echo "Unable to connect to " +_
        Circuit.Telnethostname +_
            ". Please contact IS."
        End 'Quit!
    End If
End Sub
```

This macro is now a little more robust, and can at least let the user know that something is wrong. You could also take another action, such as trying a different host name, switching to the IP address, and so forth.

Another improvement might be to observe that all of the host connection macros attached to the buttons are identical except for the host name and (potentially) the command required to log off. To streamline the button macros and centralize the connection macro, you can take advantage of the organization of SmarTerm macros into a collective. You can put the host-specific information in each button macro, and then call a single host connection macro stored in the user macro file. Try this:

1. Use Tools>Macros to create a macro in the user macro file that will do the actual connecting. It might look like this:

```
Sub ConnectToHost Hostname$
! Use this macro to connect to the host specified with Hostname$
' The actual hostname is passed in from the button macro.

If Circuit.Connected = True Then 'Are we connected?
    If Circuit.TelnetHostname = Hostname$ Then
        End 'Already connected to target host--quit!
    Else
        Session.Send LogoutCommand$ 'log off other host
        'Unable to disconnect?
        If Circuit.Disconnect = False Then
            Session.Echo "Unable to disconnect from " +_
                Circuit.Telnethostname + ". Please contact IS."
            End 'Quit!
        End If
    End If
End If
    Circuit.Telnet.Hostname = Hostname$
    If Circuit.Connect = False Then ' Unable to connect?
        Session.Echo "Unable to connect to " + Hostname$ +_
            ". Please contact IS."
        End 'Quit!
    End If
End Sub
```

2. At the top of the macro, add a public string variable that will hold the logout command for the previous host:

```
Public LogoutCommand As String

Sub ConnectToHost Hostname$
.
.
.
End Sub
```

3. Save the macro. Then use Tools>SmarTerm Buttons to create one button for each host. Attach the following macro to each button:

```
Public LogoutCommand As String

Sub Connect_ThisHost
' This macro sets the public variable LogoutCommand$ to "quit"
' (which is used when the next host is connected to) and
' connects to ThisHost.com using the common macro ConnectToHost.

LogoutCommand$ = "quit"
ConnectToHost "ThisHost.com"

End Sub
```

   As before, for each button, substitute the name of the new host for the sample text "ThisHost" and "ThisHost.com". You may also need to change the logout command.

4. Save the macros and the buttons.

   You have now streamlined the macro in each button, which merely supply a little data to the central ConnectToHost macro. If you now wanted to further improve the connection macro by adding more error-checking, starting or stopping a logfile, and so on, you need only change the ConnectToHost macro in one place, rather than in each button macro.

Sending and receiving data

The SmarTerm macro language handles all transfer of data between the host and SmarTerm, whether text or files or keystrokes, with the `Session` object and the `Transfer` object. Use the `Transfer` object for file transfer using one of the file transfer protocols SmarTerm supports (such as FTP, IND$FILE, Kermit, XMODEM, YMODEM, or ZMODEM). Use the `Session` object to send and receive keystrokes, to transfer text, and to read or write data directly to or from the terminal screen.

Note The `Session` and `Transfer` objects are those associated with the active session. If you have multiple sessions available, you should make sure that the correct one is active before sending data to the host.

Sending and receiving strings and keystrokes

There are two ways to send strings and keystrokes via a script to the host, one for text-based session types and one for form-based session types. If you are using a text-based session type such as Digital VT, Digital VT Graphics, Data General Dasher, ANSI, SCO ANSI, or Wyse, you embed the keystrokes in a string and use the `Session.Send` or `Session.SendLiteral` method. If you are using a form-based session type such as IBM 3270 or IBM 5250, you use the `Session.Sendkey` method, specifying the key with a special mnemonic.

Using Session.Send and Session.SendLiteral

The **Session.Send** and **Session.SendLiteral** commands are really quite simple. All you need to do is pass the string that you want sent to the host (or the screen, if the host is currently offline) to the **Session** object. For example, to send your username to a login prompt (as is done by the **Session_Connect** macro), you use the following command:

```
Session.Send "nguyenp" + chr(13)
```

This sends the text **"ngyuenp"** to the host, followed by a carriage return (ASCII character number 13). You can also specify the carriage-return right in the string with the built-in mnemonic **"<CR>"**:

```
Session.Send "nguyenp<CR><LF>"
```

However, you cannot use built-in mnemonics for macro commands that do not relate to SmarTerm objects. So, for example, you can assign the string to a string variable or string constant, and then pass that variable or constant to the session:

```
Dim StringToSend As String
.
.
.
StringToSend = "nguyenp<CR><LF>"
Session.Send StringToSend
```

But you cannot then use that string variable or constant with a macro command that does not relate to a SmarTerm object, such as in a dialog definition.

When you use the **Session.Send** command, SmarTerm takes the string you specify, converts any control characters you may have included to the form appropriate to the host connection (7-bit controls or 8-bit controls), and performs any character translation that you may have set with the Properties>Session Options>Character Translation tab. If you want to skip the character translation step for some reason, use the **Session.SendLiteral** command. This command, which otherwise works exactly like the **Session.Send** command, performs any 7-bit to 8-bit conversion but skips the character translation step.

Using Session.Sendkey

The **Session.Sendkey** command (only supported for form-based session types such as IBM 3270 and IBM 5250) allows you to send specific host keystrokes using standard mnemonics. These mnemonics are listed in the online help for the command. For example, you can send a down arrow keystroke with the following command:

```
Session.Sendkey "CURSORDOWN"
```

Note that, even though you use a standard mnemonic, the **Session.SendKey** command still requires you to form the keystroke into a string. This allows you to chain keystrokes together for more complicated procedures:

```
Session.Sendkey "CURSORDOWN" + "DELETEWORD" + "ENTER"
```

And, as with the **Session.Send** command, you can build the string elsewhere in the macro, assign it to a variable or constant, and then pass that variable or constant on to the command:

```
Dim KeysToSend As String
.
.
.
KeysToSend = "CURSORDOWN" + "DELETEWORD" + "ENTER"
Session.Sendkey KeysToSend
```

But you cannot then use that string variable or constant with a macro command that does not relate to a SmarTerm object, such as in a dialog definition.

Transferring text

The SmarTerm Macro Language provides a number of commands that allow you to move text back and forth between SmarTerm and a text-based host. With the SmarTerm **Session** object you can paste text to the host from a file on SmarTerm and capture text from the host into a file on SmarTerm .

Note If you routinely transfer large ASCII text files between SmarTerm and a host and you want to automate that process, you should consider using one of the file transfer protocols, such as FTP, Kermit, XMODEM, and so forth. These protocols provide extra security for your data, as they can detect and correct transmission errors and generally have a much higher throughput than straight ASCII text transfer. See the next section for information on using macros for protocol-based file transfer.

Transferring text from the host to SmarTerm

There are three ways to transfer text from the host to SmarTerm:

- Start up a text display command on the host and then use the Session.Capture command to save everything the host sends in a file on SmarTerm .
- If the information is already on the screen, use the **Session.ScreenToFile** command to put a snapshot of the text in the session window in a file on SmarTerm .
- Use the **Session.Collect** object to collect text from the host into an array of strings, and then use file-handling commands to save the strings in a file. In this section we cover only the first option, **Screen.Capture**. The second option, **Session.ScreenToFile**, is fully documented in the online help. For the third option, **Session.Collect**, see “Collect” on page 29.

There are three **Session.Capture** commands:

- **Session.CaptureFileHandling**, which lets you set whether the PC file will be replaced, or appended to
- **Session.Capture**, which starts a capture procedure
- **Session.CaptureEnd**, which ends the procedure

To use these commands properly, you also need to know the commands your host uses to display text files. In the following example, we set up the capture file handling, then capture a text file on a Digital VMS host to a file on the PC.

```
Sub CaptureHostFile
'! Capture the host file LOGIN.COM to the PC file VMSLOGIN.TXT

' First, make sure that any new capture will overwrite
' the old one
    Session.CaptureFileHandling = 0
    ' Actually, this is the default
' Now set up a LockStep object so everything stays in sync
    Dim LockStep As Object
    Set LockStep = Session.LockStep
    LockStep.Start

'Now, start up the capture
    Session.Capture("c:\vmslogin.txt")

' Now, display the host file
    Session.Send "TYPE LOGIN.COM"

' When the TYPE command is done, end the capture and
' close the file
    Session.EndCapture

' Don't forget to destroy the LockStep object!
Set LockStep = Nothing

End Sub
```

Transferring text from the SmarTerm server to the host

There are two ways in which to send text to the host:

- Use the `Session.Send` command (see “Session_Connect macro” on page 33) send individual strings to the host.
- Use the `Session.TransmitFile` command to send an ASCII text file to the host, displaying it in the session window as it does so. To use this command properly, you need to know the host commands for creating a text file, or those for starting a host application if you want to paste the text into a file.

The following sample code provides a simple example using the VMS CREATE command.

```
Sub TransmitToHost
'! Send the PC file AUTOEXEC.BAT to the host file PCAUTO.TXT

' First, set up a LockStep object so everything stays in sync
    Dim LockStep As Object
    Set LockStep = Session.LockStep
    LockStep.Start

'Now, create the file on the host
    Session.Send "CREATE PCAUTO.TXT<CR>"
```

```
' Wait a moment for the host to do its work
    Sleep 2000

' Now, display the host file
    If Session.Transmit("c:\autoexec.bat") = True Then
        Session.Send "<^Z>"    'All done--close the host file
        Session.Send "File transmitted."
    Else
        Session.Send "<^Y>"    'Error--Cancel the file creation
        Session.Send "Unable to create file."
    End If

' Don't forget to destroy the LockStep object!
Set LockStep = Nothing

End Sub
```

Transferring files

The previous section explained how to use the **`Session`** object to move text between SmarTerm and a host. You can also move other kinds of files with these methods, but it is safer to use the **`Transfer`** object. This section explains how to use the **`Transfer`** object to move files between SmarTerm and a host.

One difference between transferring text and transferring files is that there are a number of file transfer protocols that may or may not be available, depending on what the host supports. Each protocol provides different features and different interfaces. The session file always has a default transfer method installed. It is probably best to make sure that the right file transfer protocol is active before trying to use it. Use a block of code like the following:

```
'Check that we are using ZMODEM, and change to if we aren't

If Transfer.ProtocolName <> "ZMODEM" Then
    If Session.TransferProtocol "ZMODEM" = False Then
        Session.Send "Unable to select ZMODEM."
        End
    End If
End If
```

Having settled which protocol you are using, you can then use it to transfer files. The details of each file transfer protocol differ from each other. However, there are two commands that work with all transfer protocols except FTP: **`Transfer.SendFile`** and **`Transfer.ReceiveFile`**. You use both commands in much the same way, the only difference being that **`Transfer.SendFile`** sends a file to the host, while **`Transfer.ReceiveFile`** receives a file from the host. The following example uses **`Transfer.SendFile`**.

```
Sub SendFileToHost
'!Sends the file AUTOEXEC.BAT to the host using ZMODEM

'Check that we are using ZMODEM, and change to if we aren't

    If Transfer.ProtocolName <> "ZMODEM" Then
        If Session.TransferProtocol "ZMODEM" = False Then
```

```
            Session.Send "Unable to select ZMODEM."
            End
        End If
    End If

' Now set up a LockStep object so everything stays in sync
    Dim LockStep As Object
    Set LockStep = Session.LockStep
    LockStep.Start

'Start ZMODEM on the host and wait for it to take effect
    Session.Send "zmodem<CR><LF>"
    sleep 2

'Now send the file
    If Transfer.SendFile("c:\autoexec.bat") = False Then
        Session.Send "Unable to transfer file."
        End
    Else
        Session.Send "File transferred."
    End If

' Don't forget to destroy the LockStep object!
Set LockStep = Nothing

End Sub
```

Compiling Macros

You can compile and save any macro file, which is then included in the collective. Compiled macros files are available to all macro collectives in a given installation of SmarTerm, and they load and run more quickly than uncompiled macros. They cannot be debugged dynamically with the macro editor, however.

Note Compiled macro files are available to *any* collective. If you use more than one session type, or regularly connect to more than one host, organize your macros carefully so that you don't accidentally call a macro for the wrong session type or host.

Follow these steps to compile a macro file:

1. Make sure that the macro file contains bug-free macros that work properly.
2. Save the macro file with a unique name that identifies the contents of the file. For example, save all of the macros used to work on Host X as **HOSTX.STM**.
3. Load the new file into the macro editor and select any of the macros in the file for editing.
4. Save the file as a compiled macro file by typing Ctrl+Shift+D (for safety's sake, there is no menu equivalent). The macro editor compiles and saves the contents of the entire macro file in a new file with the same name but with the file extension **.PCD**. For example, the filename **HOSTX.STM** becomes **HOSTX.PCD**.

SmarTerm saves the compiled macro file in the same folder as the source macro file, usually the **\MACROS** folder. To use the new file, move (or copy) it to the SmarTerm program folder without changing the name.

Note SmarTerm will only find and use compiled macro files if they use the **.PCD** file extension and reside in the SmarTerm program folder.

Using compiled macros

When SmarTerm starts up, it looks for **.PCD** files in its program directory, loading any it finds. All the macros in the compiled files are then automatically available to macro collectives for all session types. You do not have to call the macros in a special way; they are simply available.

Symbols

' (single quote)

Syntax `'text`

Description Causes the compiler to skip all characters between this character and the end of the current line.

Example

```
Sub Main
  'This whole line is treated as a comment.
  i$="Strings"  'This is a valid assignment with a comment.
  This line will cause an error (the apostrophe is missing).
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Macro Control and Compilation on page 10

'! (description comment)

Syntax `’! text`

Description When used at the very top of a subroutine macro, causes the macro name to appear in the Tools>Macros dialog. Any text following the **’!** appears in the Description box on the Tools>Macros dialog. A macro can have up to three lines beginning with **’!** as long as they are at the very top of the macro.

Note Functions never appear in the Tools>Macro dialog, even if they begin with description comments.

Example

```
Sub Main
  '!This line appears in the Tools>Macro dialog.
  ’!So does this line.
   ’!As does this line.
  ’!This line will not appear in the dialog
  i$="This descriptive macro is now over."
  MsgBox i$
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Macro Control and Compilation on page 10

- (subtraction)

Syntax 1 **`expression1 - expression2`**

Syntax 2 **`-expression`**

Description Returns the difference between **`expression1`** and **`expression2`** or, in the second syntax, returns the negation of **`expression`**.

`expression1 - expression2`

The type of the result is the same as that of the most precise expression, with the following exceptions:

Expression One	Expression Two	Result
Long	Single	Double
Boolean	Boolean	Integer

A runtime error is generated if the result overflows its legal range.

When either or both expressions are variant, the following additional rules apply:

- If either expression is **`Null`**, then the result is **`Null`**.
- **`Empty`** is treated as an **`Integer`** of value 0.
- If the type of the result is an **`Integer`** variant that overflows, then the result is a **`Long`** variant.
- If the type of the result is a **`Long`**, **`Single`**, or **`Date`** variant that overflows, then the result is a **`Double`** variant.

`-expression`

If **`expression`** is numeric, then the type of the result is the same type as **`expression`**. If **`expression`** is **`Boolean`**, then the result is **`Integer`**.

Note In 2's complement arithmetic, unary minus may result in an overflow with **`Integer`** and **`Long`** variables when the value of **`expression`** is the largest negative number representable for that data type. For example, the following generates an overflow error:

```
Sub Main()
  Dim a As Integer
  a = -32768
  a = -a    'Generates overflow here.
End Sub
```

When negating variants, overflow will never occur because the result will be automatically promoted: integers to longs and longs to doubles.

Example

```
Sub Main
  i% = 100
  j# = 22.55
  k# = i% - j#
  Session.Echo "The difference is: " & k#
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

#Const

Syntax **`#Const constname = expression`**

Description Defines a preprocessor constant for use in the **`#If...Then...#Else`** statement. Internally, all preprocessor constants are of type **`Variant`**. Thus, the **`expression`** parameter can be any type. Variables defined using **`#Const`** can only be used within the **`#If...Then...#Else`** statement and other **`#Const`** statements. Use the **`#Const`** statement to define constants that can be used within your code.

Example

```
#Const SUBPLATFORM = "NT"
#Const MANUFACTURER = "Windows"
#Const TYPE = "Workstation"
#Const PLATFORM = MANUFACTURER & " " & SUBPLATFORM & " " & TYPE
Sub Main
  #If PLATFORM = "Windows NT Workstation" Then
    Session.Echo "Running under Windows NT Workstation"
  #End If
End Sub
```

See Also Macro Control and Compilation on page 10

#If...Then...#Else

Syntax

```
#If expression Then
[statements]
[#ElseIf expression Then
  [statements]]
[#Else
  [statements]]
#End If
```

Description Causes the compiler to include or exclude sections of code based on conditions. The **`expression`** represents any valid boolean expression evaluating to **`True`** of **`False`**. The **`expression`** may consist of literals, operators, constants defined with **`#Const`**, and any of the following predefined constants:

Constant	Value
Win32	**True**
Empty	**Empty**
False	**False**
Null	**Null**
True	**True**

The expression can use any of the following operators: **+, -, *, /, \, ^, + (unary), - (unary), Mod, &, =, <>, >=, >, <=, <, And, Or, Xor, Imp, Eqv.**

If the expression evaluates to a numeric value, then it is considered True if non-zero, False if zero. If the expression evaluates to **String** not convertible to a number or evaluates to null, then a "Type mismatch" error is generated.

Text comparisons within **expression** are always case-insensitive, regardless of the Option Compare setting

You can define your own constants using the **#Const** directive, and test for these constants within the **expression** parameter as shown below:

```
#Const VERSION = 2
Sub Main
  #If VERSION = 1 Then
    directory$ = "\apps\widget"
  #ElseIf VERSION = 2 Then
    directory$ = "\apps\widget32"
  #Else
    Session.Echo "Unknown version."
  #End If
End Sub
```

Any constant not already defined evaluates to **Empty**.

A common use of the **#If...Then...#Else** directive is to optionally include debugging statements in your code. The following example shows how debugging code can be conditionally included to check parameters to a function:

```
#Const DEBUG = 1
Sub ChangeFormat(NewFormat As Integer,StatusText As String)
  #If DEBUG = 1 Then
    If NewFormat <> 1 And NewFormat <> 2 Then
      Session.Echo "Parameter ""NewFormat"" is invalid."
      Exit Sub
    End If
    If Len(StatusText) > 78 Then
      Session.Echo "Parameter ""StatusText"" is too long."
      Exit Sub
    End If
```

```
  #End If
  Rem Change the format here...
End Sub
```

Excluded section are not compiled, allowing you to exclude sections of code that have errors or don't even represent valid syntax. For example, the following code uses the **`#If...Then...#Else`** statement to include a multi-line comment:

```
Sub Main
  #If 0
    The following section of code causes the host to display the
    first line of a famous poem:
  #End If
  Session.Echo "Don't let that horse eat that violin"
End Sub
```

In the above example, since the expression **`#If 0`** never evaluates to True, the text between that and the matching **`#End If`** will never be compiled.

Example

```
#If Win32 Then
  Declare Sub GetWindowsDirectory Lib "KERNEL32" Alias _
    "GetWindowsDirectoryA" (ByVal DirName As String,ByVal _
    MaxLen As Long)
#End If

Sub Main
  Dim DirName As String * 256
  GetWindowsDirectory DirName,len(DirName)
  Session.Echo "Windows directory = " & DirName
End Sub
```

See Also Macro Control and Compilation on page 10

& (concatenation)

Syntax **`expression1 & expression2`**

Description Returns the concatenation of **`expression1`** and **`expression2`**. If both expressions are strings, then the type of the result is **`String`**. Otherwise, the type of the result is a **`String`** variant. When nonstring expressions are encountered, each expression is converted to a **`String`** variant. If both expressions are **`Null`**, then a **`Null`** variant is returned. If only one expression is **`Null`**, then it is treated as a zero-length string. **`Empty`** variants are also treated as zero-length strings.

Note In many instances, the plus (**`+`**) operator can be used in place of **`&`**. The difference is that **`+`** attempts addition when used with at least one numeric expression, whereas **`&`** always concatenates.

Example

```
Sub Main
  s$ = "This string" & " is concatenated"
  s2$ = " with the & operator."
  Session.Echo s$ & s2$
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Character and String Manipulation on page 3.

( ) (precedence)

Syntax 1 `...(expression)...`

Syntax 2 `...,(parameter),...Description`

Parentheses override the normal precedence order of operators, forcing a subexpression to be evaluated before other parts of the expression. For example, the use of parentheses in the following expressions causes different results:

```
i = 1 + 2 * 3            'Assigns 7.

i = (1 + 2) * 3          'Assigns 9.
```

Use parentheses to make your code easier to read, removing any ambiguity in complicated expressions. You can also use parentheses when passing parameters to functions or subroutines to force a given parameter to be passed by value:

```
ShowForm i            'Pass i by reference.

ShowForm (i)            'Pass i by value.
```

Enclosing parameters within parentheses can be misleading. For example, the following statement appears to be calling a function called **`ShowForm`** without assigning the result:

```
ShowForm(i)
```

The above statement actually calls a subroutine called **`ShowForm`**, passing it the variable **`i`** by value. It may be clearer to use the **`ByVal`** keyword in this case, which accomplishes the same thing:

```
ShowForm ByVal i
```

Note The result of an expression is always passed by value.

Example

```
Sub Main
  bill = False
  dave = True
  jim = True
  If (dave And bill) Or (jim And bill) Then
      Session.Echo "The required parties for the meeting are here."
  Else
    Session.Echo "Someone is late again!"
  End If

End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Macro Control and Compilation on page 10

* (multiplication)

Syntax `expression1 * expression2`

Description Returns the product of **expression1** and **expression2**. The result is the same type as the most precise expression, with the following exceptions:

Expression One	Expression Two	Result
Single	Long	Double
Boolean	Boolean	Integer
Date	Date	Double

When the ***** operator is used with variants, the following additional rules apply:

- **Empty** is treated as 0.
- If the type of the result is an **Integer** variant that overflows, then the result is automatically promoted to a **Long** variant.
- If the type of the result is a **Single**, **Long**, or **Date** variant that overflows, then the result is automatically promoted to a **Double** variant.
- If either expression is **Null**, then the result is **Null**.

Example

```
Sub Main
  s# = 123.55
  t# = 2.55
  u# = s# * t#
  Session.Echo s# & " * " & t# & " = " & u#
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Numeric, Math, and Accounting Functions on page 9

. (dot)

Syntax 1 `object.property`

Syntax 2 `structure.member`

Description Separates an object from a property or a structure from a structure member.

Examples Use the period to separate an object from a property.

```
Sub Main
  Session.Echo Clipboard.GetText()
End Sub
```

Use the period to separate a structure from a member.

```
Type Rect
  left As Integer
  top As Integer
  right As Integer
  bottom As Integer
End Type

Sub Main
  Dim r As Rect
  r.left = 10
  r.right = 12
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Objects on page 18.

/* and */ (C-style comment block)

Syntax

```
/* text
.
.
.
*/
```

Description Causes the compiler to skip all characters between the **/*** pair and the ***/** pair.

Example

```
Sub Main
  /* This is the beginning of the comment block.
     nothing you read here will have any effect on the macro
And it doesn't matter where the text appears, until
         the appearance of the second pair: */
  i$="The comment block is done"  'This is a valid assignment.
  MsgBox i$
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Macro Control and Compilation on page 10

/ (division)

Syntax `expression1 / expression2`

Description Returns the quotient of **expression1** and **expression2**. The type of the result is **Double**, with the following exceptions:

Expression One	Expression Two	Result
Integer	Integer	Single
Single	Single	Single
Boolean	Boolean	Single

A runtime error is generated if the result overflows its legal range.

When either or both expressions is variant, then the following additional rules apply:

- If either expression is **`Null`**, then the result is **`Null`**.
- **`Empty`** is treated as an **`Integer`** of value 0.
- If both expressions are either **`Integer`** or **`Single`** variants and the result overflows, then the result is automatically promoted to a **`Double`** variant.

Example

```
Sub Main
  i% = 100
  j# = 22.55
  k# = i% / j#
  Session.Echo "The quotient of i/j is: " & k#
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Numeric, Math, and Accounting Functions on page 9

\ (integer division)

Syntax **`expression1 \ expression2`**

Description Returns the integer division of **`expression1`** and **`expression2`**. Before the integer division is performed, each expression is converted to the data type of the most precise expression. If the type of the expressions is either **`Single`**, **`Double`**, **`Date`**, or **`Currency`**, then each is rounded to **`Long`**.

If either expression is a **`Variant`**, then the following additional rules apply:

- If either expression is **`Null`**, then the result is **`Null`**.
- **`Empty`** is treated as an **`Integer`** of value 0.

Example

```
Sub Main
  s% = 100.99 \ 2.6
  Session.Echo "Integer division of 100.99\2.6 is: " & s%
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Numeric, Math, and Accounting Functions on page 9.

^ (exponentiation)

Syntax **`expression1 ^ expression2`**

Description Returns **`expression1`** raised to the power specified in **`expression2`**. The following are special cases:

Case	Value
n^0	1
0^-n	Undefined
0^+n	0
1^n	1

The type of the result is always double, except with **`Boolean`** expressions, in which case the result is **`Boolean`**. Fractional and negative exponents are allowed.

If either expression is a **`Variant`** containing **`Null`**, then the result is **`Null`**.

It is important to note that raising a number to a negative exponent produces a fractional result.

Example

```
Sub Main
  s# = 2 ^ 5          'Returns 2 to the 5th power.
  r# = 16 ^ .5        'Returns the square root of 16.
  Session.Echo "2 to the 5th power is: " & s#
  Session.Echo "The square root of 16 is: " & r#
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Numeric, Math, and Accounting Functions on page 9.

_ (line continuation)

Syntax

```
text1 _
text2
```

Description The line-continuation character, which allows you to split a single statement onto more than one line. You cannot use the line-continuation character within strings and must precede it with white space (either a space or a tab). You can follow the line-continuation character with a comment:

```
i = 5 + 6 & _        'Continue on the next line.
  "Hello"
```

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  'The line-continuation operator is useful when concatenating
  'long strings.
  mg = "This line is a line of text that" + crlf + "extends"  _
      + "beyond the borders of the editor" + crlf + "so it" _
      + "is split into multiple lines"
  'It is also useful for separating and continuing long
  'calculation lines.
  b# = .124
    a# = .223
    s# = ( (((Sin(b#) ^ 2) + (Cos(a#) ^ 2)) ^ .5) / _
```

```
        (((Sin(a#) ^ 2) + (Cos(b#) ^ 2)) ^ .5) ) * 2.00
    Session.Echo mg & crlf & "The value of s# is: " & s#
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Character and String Manipulation on page 3.

+ (addition/concatenation)

Syntax **`expression1 + expression2`**

Description Adds or concatenates two expressions. Addition operates differently depending on the type of the two expressions:

Expression One	Expression Two	Result
Numeric	Numeric	Perform a numeric add.
String	String	Concatenate, returning a string.
Numeric	String	A runtime error is generated.
Variant	String	Concatenate, returning a string variant.
Variant	Numeric	Perform a variant add.
Empty variant	Empty variant	Return an integer variant, value 0.
Empty variant	Any data type	Return the non-empty operand unchanged.
Null variant	Any data type	Return null.
Variant	Variant	Add if either is numeric; otherwise, concatenate.

When using **`+`** to concatenate two variants, the result depends on the types of each variant at runtime. You can remove any ambiguity by using the **`&`** operator.

Numeric add

A numeric add is performed when both expressions are numeric (i.e., not variant or string). The result is the same type as the most precise expression, with the following exceptions:

Expression One	Expression Two	Result
Single	Long	Double
Boolean	Boolean	Integer

A runtime error is generated if the result overflows its legal range.

Variant add

If both expressions are variants, or one expression is `Numeric` and the other expression is `Variant`, then a variant add is performed. The rules for variant add are the same as those for normal numeric add, with the following exceptions:

- If the type of the result is an `Integer` variant that overflows, then the result is a `Long` variant.
- If the type of the result is a `Long`, `Single`, or `Date` variant that overflows, then the result is a `Double` variant.

Example

```
Sub Main
  i$ = "Concatenation" + " is fun!"
  j% = 120 + 5       'Addition of numeric literals
  k# = j% + 2.7       'Addition of numeric variable
  Session.Echo "This concatenation becomes: '" i$ + _
    Str(j%) + Str(k#) & "'"
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Numeric, Math, and Accounting Functions on page 9; Character and String Manipulation on page 3.

<, <=, <>, =, >, >= (comparison)

See Comparison Operators (topic); Keywords, Data Types, Operators, and Expressions on page 6.

= (assignment)

Syntax `variable = expression`

Description Assigns the result of an expression to a variable. When assigning expressions to variables, internal type conversions are performed automatically between any two numeric quantities. Thus, you can freely assign numeric quantities without regard to type conversions. However, it is possible for an overflow error to occur when converting from larger to smaller types. This occurs when the larger type contains a numeric quantity that cannot be represented by the smaller type. For example, the following code will produce a runtime error:

```
Dim amount As Long
Dim quantity As Integer
amount = 400123          'Assign a value out of range for int.
quantity = amount          'Attempt to assign to Integer.
```

When performing an automatic data conversion, underflow is not an error.

Note The assignment operator (`=`) cannot be used to assign objects. Use the `Set` statement instead.

Example

```
Sub Main
  a$ = "This is a string"
  b% = 100
```

```
  c# = 1213.3443
  Session.Echo a$ & "," & b% & "," & c#
End Sub
```

See Also Macro Control and Compilation on page 10

A

Abs

Syntax **`Abs(expression)`**

Description Returns the absolute value of **`expression`**. If **`expression`** is **`Null`**, then **`Null`** is returned. **`Empty`** is treated as 0. The type of the result is the same as that of **`expression`**, with the following exceptions:

- If **`expression`** is an **`Integer`** that overflows its legal range, then the result is returned as a **`Long`**. This only occurs with the largest negative **`Integer`**:

```
Dim a As Variant
Dim i As Integer
i = -32768
a = Abs(i)                'Result is a Long.
i = Abs(i)                'Overflow·!
```

- If **`expression`** is a **`Long`** that overflows its legal range, then the result is returned as a **`Double`**. This only occurs with the largest negative **`Long`**:

```
Dim a As Variant
Dim l As Long
l = -2147483648
a = Abs(l)                'Result is a Double.
l = Abs(l)                'Overflow!
```

- If **`expression`** is a **`Currency`** value that overflows its legal range, an overflow error is generated.

Example

```
Sub Main
  s1% = Abs(-10.55)
  s2& = Abs(-10.55)
  s3! = Abs(-10.55)
  s4# = Abs(-10.55)
  Session.Echo "The absolute values are: " & s1% & "," & s2& & "," & s3! & ","_
& s4#
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

And

Syntax `result = expression1 And expression2`

Description Performs a logical or binary conjunction on two expressions. If both expressions are either **`Boolean`**, **`Boolean`** variants, or **`Null`** variants, then a logical conjunction is performed as follows:

Expression One	Expression Two	Result
True	True	True
True	False	False
True	Null	Null
False	True	False
False	False	False
False	Null	Null
Null	True	Null
Null	False	False
Null	Null	Null

Binary conjunction

If the two expressions are **`Integer`**, then a binary conjunction is performed, returning an **`Integer`** result. All other numeric types (including **`Empty`** variants) are converted to **`Long`**, and a binary conjunction is then performed, returning a **`Long`** result.

Binary conjunction forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions according to the following table:

Bit in Expression One	Bit in Expression Two	Result
1	1	1
0	1	0
1	0	0
0	0	0

Examples

```
Sub Main
n1 = 1001
n2 = 1000
b1 = True
b2 = False

'Perform a numeric bitwise And and store the result in N3.
n3 = n1 And n2

'Performs a logical And on B1 and B2.
If b1 And b2 Then
   Session.Echo "b1 and b2 are True; n3 is: " & n3
```

```
Else
   Session.Echo "b1 and b2 are False; n3 is: " & n3
End If

End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

AnswerBox

Syntax **`AnswerBox(prompt [,[button1] [,[button2] [,[button3] [,[title] [,helpfile,context]]]]]]])`**

Description Displays a dialog prompting the user for a response and returns an Integer indicating which button was clicked (1 for the first button, 2 for the second, and so on).AnswerBox takes the following parameters:

Parameter	Description
`prompt`	Text to be displayed above the text box. The prompt parameter can be any expression convertible to a string. The compiler resizes the dialog to hold the entire contents of prompt, up to a maximum width of 5/8 of the width of the screen and a maximum height of 5/8 of the height of the screen. The compiler word-wraps any lines too long to fit within the dialog and truncates all lines beyond the maximum number of lines that fit in the dialog. You can insert a carriage-return/line-feed character in a string to cause a line break in your message. A runtime error is generated if this parameter is null.
`button1`	The text for the first button. If omitted, then "OK and "Cancel" are used. A runtime error is generated if this parameter is null.
`button2`	The text for the second button. A runtime error is generated if this parameter is null.
`button3`	The text for the third button. A runtime error is generated if this parameter is null.
`title`	String specifying the title of the dialog. If missing, then the default title is used.
`helpfile`	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then **`context`** must also be specified.
`context`	Number specifying the ID of the topic within **`helpfile`** for this dialog's help. If this parameter is specified, then **`helpfile`** must also be specified.

The width of each button is determined by the width of the widest button.

The **`AnswerBox`** function returns 0 if the user selects Cancel.

If both the **`helpfile`** and **`context`** parameters are specified, then context-sensitive help can be invoked using the help key F1. Invoking help does not remove the dialog.

Example Display a dialog containing three buttons. Display an additional message based on which of the three buttons is selected.

```
Sub Main
  r% = AnswerBox("Copy files?", "Save", "Restore", "Cancel")
  Select Case r%
    Case 1
      Session.Echo "Files will be saved."
    Case 2
      Session.Echo "Files will be restored."
    Case Else
      Session.Echo "Operation canceled."
  End Select
End Sub
```

See Also User Interaction on page 16

Any (data type)

Description Use with the **Declare** statement to indicate that type checking is not to be performed with a given argument. For example, given the following declaration:

```
Declare Sub Foo Lib "FOO.DLL" (a As Any)
```

the following calls are valid:

```
Foo 10
Foo "Hello, world."
```

Example Call **FindWindow** to determine whether Program Manager is running. This example uses the **Any** keyword to pass a **NULL** pointer, which is accepted by the **FindWindow** function.

```
Declare Function FindWindow32 Lib "user32" Alias "FindWindowA" _
(ByVal Class As Any,ByVal Title As Any) As Long

Sub Main
  Dim hWnd As Variant
    hWnd = FindWindow32("PROGMAN",0&)
  If hWnd <> 0 Then
    Session.Echo "Program manager is running, window handle is " & hWnd
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

AppActivate

Syntax `AppActivate title | taskID,[wait]`

Description Activates an application given its name or task ID. The **AppActivate** statement takes the following named parameters:

Parameter	Description
`title`	A string containing the name of the application to be activated.
`taskID`	A number specifying the task ID of the application to be activated. Acceptable task IDs are returned by the Shell function.
`wait`	An optional boolean value indicating whether the compiler will wait for calling application to be activated before activating the specified application. If False (the default), then the compiler will activate the specified application immediately.

Note When activating applications using the task ID, it is important to declare the variable used to hold the task ID as a **`Variant`**.

Applications don't always activate immediately. To compensate, the **`AppActivate`** statement will wait a maximum of 10 seconds before failing, giving the activated application plenty of time to become activated.

The **`title`** parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches **`title`**, then a second search is performed for applications whose title string begins with **`title`**. If more than one application is found that matches **`title`**, then the first application encountered is used.

Minimized applications are not restored before activation. Thus, activating a minimized DOS application will not restore it; rather, it will highlight its icon.

A runtime error results if the window being activated is not enabled, as is the case if that application is currently displaying a modal dialog.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **`title`** parameter.

Examples Activate the Calculator.

```
Sub Main
  AppActivate "Calculator"
End Sub
```

Run another application, then activate it.

```
Sub Main
  Dim id as variant
  id = Shell("Notepad",7)      'Run Notepad minimized.
  AppActivate "Calculator"      'Activate Calculator.
  AppActivate id          'Now activate Notepad.
End Sub
```

See Also Operating System Control on page 15

AppClose

Syntax **`AppClose [title | taskID]`**

Description Closes the named application.

The **`title`** parameter is a **`String`** containing the name of the application. If the **`title`** parameter is absent, then the **`AppClose`** statement closes the active application. Or, you can specify the ID of the task as returned by the **`Shell`** function.

A runtime error results if the application being closed is not enabled, as is the case if that application is currently displaying a modal dialog.

The **`title`** parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches **`title`**, then a second search is performed for applications whose title string begins with **`title`**. If more than one application is found that matches **`title`**, then the first application encountered is used.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **`title`** parameter.

Example

```
Sub Main
  If AppFind$("Microsoft Excel") = "" Then
    Session.Echo "Excel is not running."
    Exit Sub
  End If
  AppActivate "Microsoft Excel"
  AppClose "Microsoft Excel"
End Sub
```

See Also Operating System Control on page 15

AppFind, AppFind$

Syntax **`AppFind[$] (title | taskID)`**

Description Returns a **`String`** containing the full name of the application matching either **`title`** or **`taskID`**.

The **`title`** parameter specifies the title of the application to find. If there is no exact match, the compiler will find an application whose title begins with **`title`**. Or, you can specify the ID of the task as returned by the **`Shell`** function.

The **AppFind$** functions returns a **String**, whereas the **AppFind** function returns a **String** variant. If the specified application cannot be found, then **AppFind$** returns a zero-length string and **AppFind** returns **Empty**. Using **AppFind** allows you detect failure when attempting to find an application with no caption (i.e., **Empty** is returned instead of a zero-length **String**).

AppFind$ is generally used to determine whether a given application is running. The following expression returns True if Microsoft Word is running:

```
AppFind$("Microsoft Word")
```

Example

```
Sub Main
  If AppFind$("Microsoft Excel") <> "" Then
    AppActivate "Microsoft Excel"
  Else
    Session.Echo "Excel is not running."
  End If
End Sub
```

See Also Operating System Control on page 15

AppGetActive$

Syntax **AppGetActive$()**

Description Returns a **String** containing the name of the application. If no application is active, the **AppGetActive$** function returns a zero-length string.

You can use **AppGetActive$** to retrieve the name of the active application. You can then use this name in calls to routines that require an application name.

Example

```
Sub Main
  n$ = AppGetActive$()
  AppMinimize n$
End Sub
```

See Also Operating System Control on page 15

AppGetPosition

Syntax **AppGetPosition x,y,width,height [,title | taskID]**

Description Retrieves the position of the named application. The **AppGetPosition** statement takes the following parameters:

Parameter	Description
x, y	Names of integer variables to receive the position of the application's window.
width, height	Names of integer variables to receive the size of the application's window.
title	A string containing the name of the application. If the **title** parameter is omitted, then the active application is used.
taskID	A number specifying the task ID of the application to be activated. Acceptable task IDs are returned by the Shell function.

The **x**, **y**, **width**, and **height** variables are filled with the position and size of the application's window. If an argument is not a variable, then the argument is ignored, as in the following example, which only retrieves the **x** and **y** parameters and ignores the **width** and **height** parameters:

```
Dim x as integer, y as integer
AppGetPosition x,y,0,0,"Program Manager"
```

The position and size of the window are returned in twips (1440^{th} parts of an inch).

The **title** parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches **title**, then a second search is performed for applications whose title string begins with **title**. If more than one application is found that matches **title**, then the first application encountered is used.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **title** parameter.

Example

```
Sub Main
  Dim x As Integer, y As Integer
  Dim cx As Integer, cy As Integer
  AppGetPosition x,y,cx,cy,"Program Manager"
End Sub
```

See Also Operating System Control on page 15

AppGetState

Syntax **AppGetState[([title | taskID])]**

Description Returns an **Integer** specifying the state of the specified top-level window. The **AppGetState** function returns any of the following values:

If Window Is	AppGetState Returns	Value
Maximized	**ebMinimized**	1
Minimized	**ebMaximized**	2
Restored	**ebRestored**	3

The **title** parameter is a **String** containing the name of the desired application. If it is omitted, then the **AppGetState** function returns the name of the active application.

Or, you can specify the ID of the task as returned by the **Shell** function.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **title** parameter.

Example

```
Sub Main
  If AppFind$("Untitled - Notepad") = "" Then
    Session.Echo "Can't find Untitled - Notepad."
    Exit Sub
  End If
  AppActivate "Untitled - Notepad"  'Activate ProgMan
  state = AppGetState        'Save its state.
  AppMinimize          'Minimize it.
  Session.Echo "Notepad is now minimized. Select OK to restore it."
  AppActivate "Untitled - Notepad"
  AppSetState state        'Restore it.
End Sub
```

See Also Operating System Control on page 15

AppHide

Syntax **AppHide [title | taskID]**

Description Hides the named application. If the named application is already hidden, the **AppHide** statement will have no effect.

The **title** parameter is a **String** containing the name of the desired application. If it is omitted, then the **AppHide** statement hides the active application. Or, you can specify the ID of the task as returned by the **Shell** function.

AppHide generates a runtime error if the named application is not enabled, as is the case if that application is displaying a modal dialog.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is

"Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **`title`** parameter.

Example

```
Sub Main
  'See whether Untitled - Notepad is running.
  If AppFind$("Untitled - Notepad") = "" Then Exit Sub
  AppHide "Untitled - Notepad"
  Session.Echo "Untitled - Notepad is now hidden. Press OK to show it once again."
  AppShow "Untitled - Notepad"
End Sub
```

See Also Operating System Control on page 15

Application (object)

The Application object provides access to aspects of SmarTerm that are global to all session types, such as the exact product name and version, the locations of the user files, and so forth.

Application.ActiveSession

Syntax **`Application.ActiveSession`**

Description Returns an object representing SmarTerm’s current session.

Example

```
Dim Active as Object
Set Active = Application.ActiveSession
```

Application.Application

Syntax **`Application.Application`**

Description Returns SmarTerm’s application object.

Example

```
Dim App as Object
Set App = Application.Application
```

See Also Application and Session Features on page 11

Application.Caption

Syntax **`Application.Caption`**

Description Returns or sets SmarTerm’s application window caption (string).

Example Return SmarTerm's main window caption and set it to "SmarTerm"

```
Sub Main
  Dim CurrentCaption as String
  CurrentCaption = Application.Caption
```

```
  Session.Echo "Current window caption is " & CurrentCaption
  Application.Caption = "SmarTerm"
End Sub
```

See Also Session.Caption; Application and Session Features on page 11

Application.CommandLine

Syntax `Application.CommandLine`

Description Returns the command line from when the application was started (string). The command line switch **"-$"** or **"/$"** causes SmarTerm to ignore all command line arguments that follow it. Additional characters can be appended to the switch (e.g., **"-$hello"**) and still be recognized. This can be useful for placing parameters on the command line that are intended for access by a macro.

Example

```
Sub Main
  Dim StCmdLine as String
  StCmdLine = Application.CommandLine
  Session.Echo "Current command line is " & StCmdLine
End Sub
```

See Also Session.Caption; Application and Session Features on page 11

Application.DoMenuFunction

Syntax `Application.DoMenuFunction menuitem$`

where **menuitem$** is the menu item to trigger (string).

Description Triggers an application-based menu action in SmarTerm.Possible values:

FileExit	PropertiesOptions
FileNew	ToolsRestoreAll
FileOpen	ToolsUndoRestore
FilePageSetup	ViewFullScreen
FileSaveWorkspace	ViewMenuBar
HelpAboutSmarTermOffice	ViewStatusBar
HelpMacroGuide	ViewToolbar
HelpSmarTermHelpTopics	ViewWorkbook
HelpTechnicalSupport	WindowArrangeIcons
HelpUserHelp	WindowCascade
PropertiesLanguage	WindowTile

Example

```
Sub Main
  Application.DoMenuFunction "ViewFullScreen"
End Sub
```

See Also Session.DoMenuFunction; Application and Session Features on page 11

Application.FlashIcon

Syntax `Application.FlashIcon`

Description Returns or sets whether SmarTerm's session icon should blink when new information is received from a host (boolean).

Example

```
Sub Main
  Dim FlashState as Boolean
  FlashState = Application.FlashIcon
  If FlashState = FALSE then
    Session.Echo "Setting SmarTerm session icon to flash"
    Application.FlashIcon = TRUE
  End If
End Sub
```

See Also Session.DoMenuFunction; Application and Session Features on page 11

Application.InstalledLanguages

Syntax `Application.InstalledLanguages(index)`

where **`index`** is the index of the language value to retrieve (integer).

Description Returns a value representing the installed language corresponding to the index value provided (integer). This function should be called initially with the index set to 1. This will return a non-zero value if a language has been retrieved. While the value returned is non-zero, increment the index by one and continue calling. This will retrieve as many languages as have been installed.

Possible values are:

Value	Constant	Meaning
1031	`smlGERMAN`	German.
1033	`smlENGLISH`	English.
1036	`smlFRENCH`	French.
1034	`smlSPANISH`	Spanish.

Example

```
Sub Main
  Dim LanguageChoices() as Integer
  Dim Continue as Boolean
  Dim i, Value as Integer
  Continue = True
  i = 1
  Do
    Value = Application.InstalledLanguages (I)
    If Value <> 0 Then
        Redim Preserve LanguageChoices(i)
        LanguageChoices(i-1) = Value
        i = i + 1
    Else
        Continue = False
```

```
    End If
  Loop While Continue = True
End Sub
```

See Also Application.StartupLanguage; Session.Language; Application and Session Features

Application.Parent

Syntax **`Application.Parent`**

Description Returns the SmarTerm application's parent object (which is always **`Nothing`**).

Example

```
Dim Parent as Object
Parent = Application.Parent
```

See Also Application and Session Features on page 11

Application.Product

Syntax **`Application.Product`**

Description Returns a string identifying the SmarTerm product in use.

Example

```
Sub Main
  Dim ProdName as String
  ProdName = Application.Product
  Session.Echo "The SmarTerm product name is " & ProdName
End Sub
```

See Also Application.Version; Application and Session Features on page 11

Application.Quit

Syntax **`Application.Quit`**

Description Terminates the SmarTerm application, including all open sessions.

Example

```
Sub Main
  Dim nMsg as integer
  nMsg = Session.Echo ("This script will stop SmarTerm. OK?",ebYesNo)
  if nMsg = ebYes then
    Application.Quit
  End If
End Sub
```

See Also Circuit.Disconnect; Application and Session Features on page 11

Application.Sessions (collection)

Syntax See specific uses of this collection.

Description Returns an object representing the collection of sessions within SmarTerm (object). The Sessions collection object supports access to all sessions running within the SmarTerm application. This object's methods and properties will be of primary use when accessing SmarTerm through an external OLE Automation controller.

Example This code is meant to be run from an external OLE Automation controller in which the Application, Session, Circuit, and Transfer objects are not predefined.

```
Dim Application As Object
Dim Session As Object
Dim Circuit As Object
Dim Transfer As Object
Dim SessionFileSpec As String
Set Application = CreateObject("SmarTerm.Application")
SessionFileSpec = Application.UserSessionsLocation & "\session1.stw"
Set Session = Application.Sessions.Open(SessionFileSpec)
Set Circuit = Session.Circuit
Set Transfer = Session.Transfer
```

This code is meant to be run from an external controller to attach to an existing SmarTerm process and locate a session captioned "MyHost".

```
Dim TotalSessions, I as Integer
Dim TestSession as Object
Dim Session As Object
Dim Circuit As Object
Dim Transfer As Object
Dim FoundMatch as Boolean
Set Application = GetObject(, "SmarTerm.Application")
TotalSessions = Application.Sessions.Count
FoundMatch = False
If TotalSessions > 0 Then
    For I = 0 to (TotalSessions - 1)
        Set TestSession = Application.Sessions.Item(I)
        If TestSession.Caption = "Session1" Then
            FoundMatch = True
            Exit For
        End If
    Next I
End If
If FoundMatch Then
    Set Session = TestSession
    Set Circuit = Session.Circuit
    Set Transfer = Session.Transfer
End If
```

Similar to above, but for the case in which the automation controller supports a 'For Each' statement that iterates through a collection.

```
Dim TestSession as Object
Dim Session As Object
Dim Circuit As Object
Dim Transfer As Object
Dim FoundMatch as Boolean
Set Application = GetObject(, "SmarTerm.Application")
TotalSessions = Application.Sessions.Count
FoundMatch = False
For Each TestSession In Application.Sessions
```

```
    If TestSession.Caption = "Session1" Then
        FoundMatch = True
        Exit For
    End If
Next
If FoundMatch Then
    Set Session = TestSession
    Set Circuit = Session.Circuit
    Set Transfer = Session.Transfer
End If
```

See Also Application and Session Features on page 11; Objects on page 18

Application.Sessions.Application

Syntax `Application.Sessions.Application`

Description Returns the SmarTerm application object.

Example

```
Dim App as Object
Set App = Application.Sessions.Application
```

See Also Application and Session Features on page 11; Objects on page 18

Application.Sessions.Count

Syntax `Application.Sessions.Count`

Description Returns an integer containing the number of sessions maintained by the Sessions collection.

Example See the examples for **`Application.Sessions`**.

See Also Application and Session Features on page 11

Application.Sessions.Item

Syntax `Application.Sessions.Item(sessionindex%)`

where **`sessionindex%`** is an integer, index of the session to access.

Description Returns a session object of the specified session ID.

Example See the examples for **`Application.Sessions`**.

See Also Application and Session Features on page 11

Application.Sessions.Open

Syntax `Application.Sessions.Open sessionfile$`

where **`sessionfile$`** is the name of the session file to open.

Description Returns a session object after opening the specified session. Returns **`Nothing`** if the method fails.

Example See the examples for **`Application.Sessions`**.

See Also Application and Session Features on page 11; Objects on page 18

Application.Sessions.Parent

Syntax **`Application.Sessions.Parent`**

Description Returns SmarTerm's parent object.

Example

```
Dim Parent as Object
Parent = Application.Sessions.Parent
```

See Also Application and Session Features on page 11; Objects on page 18

Application.StartupLanguage

Syntax **`Application.StartupLanguage`**

Description Returns the startup language that was selected during Setup (integer). Possible values are:

Value	Constant	Meaning
1031	**`smlGERMAN`**	German.
1033	**`smlENGLISH`**	English.
1036	**`smlFRENCH`**	French.
1034	**`smlSPANISH`**	Spanish.

Example Report an error in the language chosen as the startup language

```
Sub Main
  Dim StartupLanguage as Integer
  StartupLanguage = Application.StartupLanugage
  Select Case StartupLanguage
    Case 1031   ' German
        Session.Echo "Ein Fehler ist aufgetreten."
    Case 1033   ' English
        Session.Echo "An error has occurred."
    Case 1036   ' French
        Session.Echo "Une erreur est survenue."
    Case 1034   ' Spanish
        Session.Echo "Ocurrió un error."
  End Select
End Sub
```

See Also Application.InstalledLanguages; Session.Language; Application and Session Features on page 11

Application.SuppressRefocus

Syntax **`Application.SuppressRefocus= true|false`**

Description Returns or sets the state of the focus when control returns to SmarTerm (Boolean). If false (the default), a macro that launches another application (such as Notepad) returns the focus to SmarTerm as soon as the macro ends. This means that, if the other application typically displays a window requiring user input, that window may be covered by SmarTerm's session window. If Application.SuppressRefocus is true, then the focus returns to SmarTerm at the end of the macro only if no other applications have been launched. This allows the other application's window to remain in the foreground until dismissed by the user.

Note Application.SuppressRefocus is always reset to FALSE when the macro ends. You must reset it to TRUE every time you wish to supress automatic refocus.

Example

```
Sub Main
'! Launches NOTEPAD.EXE and lets it keep focus.
 Dim TaskID As Variant
 TaskID = Shell("notepad", ebNormalFocus)
 Application.SuppressReFocus TRUE
End Sub
```

See Also Application and Session Features on page 11; User Interaction on page 16

Application.UserHelpFile

Syntax **`Application.UserHelpFile`**

Description Returns or sets the name of the SmarTerm user help file (string).

Example

```
Sub Main
   Dim HelpFile as String
   HelpFile = Application.UserHelpFile
   Session.Echo "Current help file was " & HelpFile
   Session.Echo "Changing help file to VAXMAIL"
   Application.UserHelpFile = "VAXMAIL.HLP"
End Sub
```

See Also Application.UserHelpMenu; Application.ViewUserHelp; Application and Session Features on page 11; User Interaction on page 16

Application.UserHelpMenu

Syntax **`Application.UserHelpMenu`**

Description Returns or sets the menu choice for SmarTerm's user help.

Example

```
Sub Main
   Dim HelpMenu as String
   HelpMenu = Application.UserHelpMenu
   Session.Echo "Current help file was " & HelpMenu
```

```
   Session.Echo "Changing help menu for VAX Mail"
   Application.UserHelpMenu = "How to use VAX Mail"
End Sub
```

See Also Application.SuppressRefocus; Application.ViewUserHelp; Application and Session Features on page 11; User Interaction on page 16

Application.UserHotSpotsLocation

Syntax `Application.UserHotSpotsLocation`

Description Returns or sets the file location for SmarTerm's user HotSpots (string).

Example

```
Sub Main
  Dim  Location as String
  Location = Application.UserHotSpotsLocation
  Application.UserHotSpotsLocation = "c:\hotspots"
End Sub
```

See Also Application and Session Features on page 11

Application.UserKeyMapsLocation

Syntax `Application.UserKeyMapsLocation`

Description Returns or sets the file location for SmarTerm's user keyboard maps (string).

Example

```
Sub Main
  Dim  Location as String
  Location = Application.UserKeyMapsLocation
  Application.UserKeyMapsLocation = "c:\keymaps"
End Sub
```

See Also Application and Session Features on page 11

Application.UserMacrosLocation

Syntax `Application.UserMacrosLocation`

Description Returns or sets the file location for SmarTerm's user macros (string).

Example

```
Sub Main
  Dim  Location as String
  Location = Application.UserMacrosLocation
  Application.UserMacrosLocation = "c:\macros"
End Sub
```

See Also Application and Session Features on page 11

Application.UserPhoneBookLocation

Syntax `Application.UserPhoneBookLocation`

Description Returns or sets the file location for SmarTerm's user phonebook (string).

Example

```
Sub Main
  Dim  Location as String
  Location = Application.UserPhoneBookLocation
  Application.UserPhoneBookLocation = "c:\phonebk"
End Sub
```

See Also Application and Session Features on page 11; Host Connections on page 7

Application.UserSessionsLocation

Syntax `Application.UserSessionsLocation`

Description Returns or sets the file location for SmarTerm's user session files (string).

Example

```
Sub Main
  Dim  Location as String
  Location = Application.UserSessionsLocation
  Application.UserSessionsLocation = "c:\sessions"
End Sub
```

See Also Application and Session Features on page 11

Application.UserButtonPicturesLocation

Syntax `Application.UserButtonPicturesLocation`

Description Returns or sets the file location for SmarTerm's user Buttons graphic files (string).

Example

```
Sub Main
  Dim  Location as String
  Location = Application.UserButtonPicturesLocation
  Application.UserButtonPicturesLocation = "c:\butnpix"
End Sub
```

See Also Application and Session Features on page 11

Application.UserSmarTermButtonsLocation

Syntax `Application.UserSmarTermButtonsLocation`

Description Returns or sets the file location for user SmarTerm Buttons files (string).

Example

```
Sub Main
  Dim  Location as String
  Location = Application.UserSmarTermButtonsLocation
  Application.UserSmarTermButtonsLocation = "c:\buttons"
End Sub
```

See Also Application and Session Features on page 11

Application.UserTransfersLocation

Syntax `Application.UserTransfersLocation`

Description Returns or sets the file location for SmarTerm file transfers.

Example

```
Sub Main
  Dim  Location as String
  Location = Application.UserTransfersLocation
  Application.UserTransfersLocation = "c:\transfer"
End Sub
```

See Also Application and Session Features on page 11

Application.Version

Syntax `Application.Version`

Description Returns a string identifying the version number of SmarTerm’s macro engine.

Example

```
Sub Main
  Dim MacroVersion as String
  MacroVersion = Application.Version
  Session.Echo "SmarTerm's macro version number is " & MacroVersion
End Sub
```

See Also Application.Product; Application and Session Features on page 11

Application.ViewUserHelp

Syntax `Application.ViewUserHelp`

Description Launches the user defined help file in the help viewer.

Example

```
Sub Main
  Application.ViewUserHelp
End Sub
```

See Also Application.SuppressRefocus; Application.UserHelpMenu; Application and Session Features on page 11; User Interaction on page 16

Application.Visible

Syntax `Application.Visible`

Description Returns or sets the visible state of the SmarTerm application (boolean). This property can be used to make SmarTerm invisible.

Example

```
Sub Main
  Dim Visible as Boolean
  Visible = Application.Visible
  If Visible = True Then
     Session.Echo "Hiding SmarTerm"
```

```
    Application.Visible = False
  End If
End Sub
```

See Also Session.Visible

Application.WindowState

Syntax **`Application.WindowState`**

Description Returns or sets the state of the SmarTerm application window (integer). Possible values are:

Value	Constant	Meaning
0	**`smlMINIMIZE`**	The window is minimized.
1	**`smlRESTORE`**	The window is restored.
2	**`smlMAXIMIZE`**	The window is maximized.

Example

```
Sub Main
  Dim WinState as Integer
  WinState = Application.WindowState
  If WinState = smlMINIMIZE Then
    Application.WindowState = smlMAXIMIZE
  End If
End Sub
```

See Also Session.WindowState; Application and Session Features on page 11

AppList

Syntax **`AppList AppNames$()`**

Description Fills an array with the names of all open applications. The **`AppNames$`** parameter must specify either a zero- or one-dimensional dynamic **`String`** array or a one-dimensional fixed **`String`** array. If the array is dynamic, then it will be redimensioned to match the number of open applications. For fixed arrays, **`AppList`** first erases each array element, then begins assigning application names to the elements in the array. If there are fewer elements than will fit in the array, then the remaining elements are unused. The compiler returns a runtime error if the array is too small to hold the new elements.

After calling this function, you can use **`LBound`** and **`UBound`** to determine the new size of the array.

Example

```
Sub Main
  AppList apps
  'Check to see whether any applications were found.
  If ArrayDims(apps) = 0 Then Exit Sub
  For i = LBound(apps) To UBound(apps)
    AppMinimize apps(i)
  Next i
End Sub
```

See Also Operating System Control on page 15

AppMaximize

Syntax **`AppMaximize [title | taskID]`**

Description Maximizes the named application.

The **`title`** parameter is a **`String`** containing the name of the desired application. If it is omitted, then the **`AppMaximize`** function maximizes the active application. Or, you can specify the ID of the task as returned by the **`Shell`** function.

If the named application is maximized or hidden, the **`AppMaximize`** statement will have no effect.

The **`title`** parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches **`title`**, then a second search is performed for applications whose title string begins with **`title`**. If more than one application is found that matches **`title`**, then the first application encountered is used.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **`title`** parameter.

`AppMaximize` generates a runtime error if the named application is not enabled, as is the case if that application is displaying a modal dialog.

Example

```
Sub Main
  AppMaximize "Untitled - Notepad"
  'Maximize Untitled - Notepad.
  If AppFind$("NotePad") <> "" Then
    AppActivate "NotePad"
  'Set the focus to NotePad.
    AppMaximize          'Maximize it.
  End If
End Sub
```

See Also Operating System Control on page 15

AppMinimize

Syntax **`AppMinimize [title | taskID]`**

Description Minimizes the named application.

The **`title`** parameter is a **`String`** containing the name of the desired application. If it is omitted, then the **`AppMinimize`** function minimizes the active application. Or, you can specify the ID of the task as returned by the **`Shell`** function.

If the named application is minimized or hidden, the **`AppMinimize`** statement will have no effect.

The **`title`** parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches **`title`**, then a second search is performed for applications whose title string begins with **`title`**. If more than one application is found that matches **`title`**, then the first application encountered is used.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **`title`** parameter.

`AppMinimize` generates a runtime error if the named application is not enabled, as is the case if that application is displaying a modal dialog.

Example

```
Sub Main
  AppMinimize "Untitled - Notepad"
  'Maximize Untitled - Notepad.
  If AppFind$("NotePad") <> "" Then
    AppActivate "NotePad"
  'Set the focus to NotePad.
    AppMinimize          'Maximize it.
  End If
End Sub
```

See Also Operating System Control on page 15

AppMove

Syntax **`AppMove x,y [,title | taskID]`**

Description Sets the upper left corner of the named application to a given location. The **`AppMove`** statement takes the following parameters:

Parameter	Description
`x, y`	Integer coordinates specifying the upper left corner of the new location of the application, relative to the upper left corner of the display.
`title`	String containing the name of the application to move. If this parameter is omitted, then the active application is moved.
`taskID`	A number specifying the task ID of the application to be activated. Acceptable task IDs are returned by the Shell function.

If the named application is maximized or hidden, the **`AppMove`** statement will have no effect.

The **`x`** and **`y`** parameters are specified in twips.

`AppMove` will accept **`x`** and **`y`** parameters that are off the screen.

The **`title`** parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches **`title`**, then a second search is performed for applications whose title string begins with **`title`**. If more than one application is found that matches **`title`**, then the first application encountered is used.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **`title`** parameter.

`AppMove` generates a runtime error if the named application is not enabled, as is the case if that application is currently displaying a modal dialog.

Example

```
Sub Main
  Dim x%,y%
  AppActivate "Untitled - Notepad"      'Activate Program Mgr.
  AppGetPosition x%,y%,0,0              'Retrieve its position.
  x% = x% + Screen.TwipsPerPixelX * 10  'Add 10 pixels.
  AppMove x% + 10,y%                    'Nudge it 10 pixels
End Sub
```

See Also Operating System Control on page 15

AppRestore

Syntax **`AppRestore [title | taskID]`**

Description Restores the named application.

The **`title`** parameter is a **`String`** containing the name of the application to restore. If this parameter is omitted, then the active application is restored. Or, you can specify the ID of the task as returned by the **`Shell`** function.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **`title`** parameter.

`AppRestore` will have an effect only if the main window of the named application is either maximized or minimized.

`AppRestore` will have no effect if the named window is hidden.

`AppRestore` generates a runtime error if the named application is not enabled, as is the case if that application is currently displaying a modal dialog.

Example

```
Sub Main
  If AppFind$("Untitled - Notepad") = "" Then Exit Sub
  AppActivate "Untitled - Notepad"
  AppMinimize "Untitled - Notepad"
  Session.Echo "Untitled - Notepad is now minimized. Press OK to restore it."
  AppRestore "Untitled - Notepad"
End Sub
```

See Also Operating System Control on page 15

AppSetState

Syntax **`AppSetState newstate [,title | taskID]`**

Description Maximizes, minimizes, or restores the named application, depending on the value of **`newstate`**. The **`AppSetState`** statement takes the following parameters:

Parameter	Description
`newstate`	An integer specifying the new state of the window.
`title`	A string containing the name of the application to change. If omitted, then the active application is used.
`taskID`	A number specifying the task ID of the application to be activated. Acceptable task IDs are returned by the Shell function.

The **`newstate`** parameter can be any of the following values:

Value	Constant	Description
1	**`ebMinimized`**	The named application is minimized.
2	**`ebMaximized`**	The named application is maximized.
3	**`ebRestored`**	The named application is restored.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **`title`** parameter.

Example See AppGetState (function).

See Also Operating System Control on page 15

AppShow

Syntax **`AppShow [title | taskID]`**

Description Makes the named application visible.

The **`title`** parameter is a **`String`** containing the name of the application to show. If this parameter is omitted, then the active application is shown. Or, you can specify the ID of the task as returned by the **`Shell`** function.

If the named application is already visible, **`AppShow`** will have no effect.

The **`title`** parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches **`title`**, then a second search is performed for applications whose title string begins with **`title`**. If more than one application is found that matches **`title`**, then the first application encountered is used.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **`title`** parameter.

`AppShow` generates a runtime error if the named application is not enabled, as is the case if that application is displaying a modal dialog.

Example See AppHide (statement).

See Also Operating System Control on page 15

AppSize

Syntax **`AppSize width,height [,title | taskID]`**

Description Sets the width and height of the named application. The **`AppSize`** statement takes the following parameters:

Parameter	Description
`width, height`	Integer coordinates specifying the new size of the application.
`title`	String containing the name of the application to resize. If this parameter is omitted, then the active application is use.
`taskID`	A number specifying the task ID of the application to be activated. Acceptable task IDs are returned by the Shell function.

The **`width`** and **`height`** parameters are specified in twips.

This statement will only work if the named application is restored (i.e., not minimized or maximized).

The **`title`** parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches **`title`**, then a second search is performed for applications whose title string begins with **title**. If more than one application is found that matches **title**, then the first application encountered is used.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **title** parameter.

A runtime error results if the application being resized is not enabled, which is the case if that application is displaying a modal dialog when an **AppSize** statement is executed.

Example

```
Sub Main
  Dim w%,h%
  AppGetPosition 0,0,w%,h%              'Get current width/height.
  x% = x% + Screen.TwipsPerPixelX * 10  'Add 10 pixels.
  y% = y% + Screen.TwipsPerPixelY * 10  'Add 10 pixels.
  AppSize w%,h%                         'Change to new size.
End Sub
```

See Also Operating System Control on page 15

AppType

Syntax **AppType [(title | taskID)]**

Description Returns an **Integer** indicating the executable file type of the named application:

Returns	If the file type is
ebDos	DOS executable
ebWindows	Windows executable

The **title** parameter is a **String** containing the name of the application. If this parameter is omitted, then the active application is used. Or, you can specify the ID of the task as returned by the **Shell** function.

Under Windows 98/Me, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 98/Me, the caption is "Untitled - Notepad". You must keep this in mind when specifying the **title** parameter.

Example This example creates an array of strings containing the names of all the running Windows applications. It uses the **AppType** command to determine whether an application is a Windows app or a DOS app.

```
Sub Main
  Dim apps$(),wapps$()
  AppList apps    'Retrieve a list of all Windows and DOS apps.
  If ArrayDims(apps) = 0 Then
```

```
    Session.Echo "There are no running applications."
    Exit Sub
  End If
  'Create an array to hold only the Windows apps.
  ReDim wapps$(UBound(apps))
  n = 0  'Copy the Windows apps from one array to the target array.
  For i = LBound(apps) to UBound(apps)
    If AppType(apps(i)) = ebWindows Then
      wapps(n) = apps(i)
      n = n + 1
    End If
  Next i
  If n = 0 Then  'Make sure at least one Windows app was found.
    Session.Echo "There are no running Windows applications."
    Exit Sub
  End If
  ReDim Preserve wapps(n - 1)   'Resize to hold the exact number.
  'Let the user pick one.
  index% = SelectBox("Windows Applications","Select a Windows application:",wapps)
End Sub
```

See Also Operating System Control on page 15

ArrayDims

Syntax **`ArrayDims(arrayvariable)`**

Description Returns an **`Integer`** indicating the number of dimensions in the array. A return value of 0 indicates that the array has not yet been dimensioned. This function can be used to determine whether a given array contains any elements or if the array is initially created with no dimensions and then redimensioned by another function, such as the **`FileList`** function, as shown in the following example.

Example This example allocates an empty (null-dimensioned) array, fills the array with a list of filenames, which resizes the array, then tests the array dimension.

```
Sub dimensions

Dim f$()
Dim message$
Dims% = Arraydims(f$)
Message$ = "The array size is "

If Dims% = 0 Then
   Session.Echo "The array is empty"
Else
   For i% = 1 To Dims%
      If i < Dims Then
         Message$ = Message$ & (Ubound(f$,i) - Lbound(f$,i)+1) & " X "
      Else
         Message$ = Message$ & (Ubound(f$,i) - Lbound(f$,i)+1)
      End If
   Next i%
   Session.Echo Message$
```

```
End If

End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Arrays (topic)

Declaring array variables

Arrays are declared using any of the following statements:

```
Dim
Public
Private
```

For example:

```
Dim a(10) As Integer
Public LastNames(1 to 5,-2 to 7) As Variant
Private
```

Arrays of any data type can be created, including **`Integer`**, **`Long`**, **`Single`**, **`Double`**, **`Boolean`**, **`Date`**, **`Variant`**, **`Object`**, user-defined structures, and data objects.

The lower and upper bounds of each array dimension must be within the following range:

```
-32768 <= bound <= 32767
```

Arrays can have up to 60 dimensions.

Arrays can be declared as either fixed or dynamic, as described below.

Fixed arrays

The dimensions of fixed arrays cannot be adjusted at execution time. Once declared, a fixed array will always require the same amount of storage. Fixed arrays can be declared with the **`Dim`**, **`Private`**, or **`Public`** statement by supplying explicit dimensions. The following example declares a fixed array of ten strings:

```
Dim a(10) As String
```

Fixed arrays can be used as members of user-defined data types. The following example shows a structure containing fixed-length arrays:

```
Type Foo
  rect(4) As Integer
  colors(10) As Integer
End Type
```

Only fixed arrays can appear within structures.

Dynamic arrays

Dynamic arrays are declared without explicit dimensions, as shown below:

```
Public Ages() As Integer
```

Dynamic arrays can be resized at execution time using the **`Redim`** statement:

```
Redim Ages$(100)
```

Subsequent to their initial declaration, dynamic arrays can be redimensioned any number of times. When redimensioning an array, the old array is first erased unless you use the **`Preserve`** keyword, as shown below:

```
Redim Preserve Ages$(100)
```

Dynamic arrays cannot be members of user-defined data types.

Passing arrays

Arrays are always passed by reference. When you pass an array, you can specify the array name by itself, or with parentheses as shown below:

```
Dim a(10) As String
FileList a          'Both of these are OK
FileList a()
```

Querying arrays

Use these functions to retrieve information about arrays:

Use this function	To
`LBound`	Retrieve the lower bound of an array. A runtime error is generated if the array has no dimensions.
`UBound`	Retrieve the upper bound of an array. A runtime error is generated if the array has no dimensions.
`ArrayDims`	Retrieve the number of dimensions of an array. This function returns 0 if the array has no dimensions.

Operations on arrays

The following table indicates the functions that operate on arrays:

Command	Action
`ArraySort`	Sort an array of integers, longs, singles, doubles, currency, **`booleans`**, dates, or variants.
`FileList`	Fill an array with a list of files in a given directory.
`DiskDrives`	Fill an array with a list of valid drive letters.
`AppList`	Fill an array with a list of running applications.
`SelectBox`	Display the contents of an array in a listbox.
`PopupMenu`	Display the contents of an array in a popup menu.
`ReadIniSection`	Fill an array with the item names from a section in an INI file.
`FileDirs`	Fill an array with a list of folders.
`Erase`	Erase all the elements of an array.
`ReDim`	Establish the bounds and dimensions of an array.
`Dim`	Declare an array.

ArraySort

Syntax **`ArraySort array()`**

Description Sorts a single-dimensioned array in ascending order. If a string array is specified, then the routine sorts alphabetically in ascending order using case-sensitive string comparisons. If a numeric array is specified, the **`ArraySort`** statement sorts smaller numbers to the lowest array index locations. There is a runtime error if you specify an array with more than one dimension.

When sorting an array of variants, the following rules apply:

- A runtime error is generated if any element of the array is an object.
- **`String`** is greater than any numeric type.
- **`Null`** is less than **`String`** and all numeric types.
- **`Empty`** is treated as a number with the value 0.
- **`String`** comparison is case-sensitive (this function is not affected by the **`Option Compare`** setting).

Example

```
Sub Main
  Dim f$()
  FileList f$,"c:\*.*"
  ArraySort f$
  Session.Echo "Files: <CR><LF>"
  For i= 0 to UBound(f$)
    Session.Echo f$(i) & "<CR><LF>"
  Next i
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6.

Asc, AscB, AscW

Syntax

```
Asc(string)
AscB(string)
AscW(string)
```

Description Returns an **`Integer`** containing the numeric code for the first character of **`string`**. On single-byte systems, this function returns a number between 0 and 255, whereas on MBCS systems, this function returns a number between -32768 and 32767. On wide platforms, this function returns the MBCS character code after converting the wide character to MBCS.

To return the value of the first byte of a string, use the **`AscB`** function. This function is used when you need the value of the first byte of a string known to contain byte data rather than character data. On single-byte systems, the **`AscB`** function is identical to the **`Asc`** function.

The **`AscW`** function returns the character value native to that platform. For example, on Win32 platforms, this function returns the UNICODE character code.

The following table summarizes the values returned by these functions:

Function	String Format	Return Value
`Asc`	SBCS	First byte of string (between 0 and 255)
	MBCS	First character of string (between -32769 and 32767)
	Wide	First character of string after conversion to MBCS.
`AscB`	SBCS	First byte of string.
	MBCS	First byte of string.
	Wide	First byte of string.
`AscW`	SBCS	Same as **`Asc`**.
	MBCS	Same as **`Asc`**.
	Wide	Wide character native to operating system.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  s$ = InputBox("Please enter a string.","Enter String")
  If s$ = "" Then End    'Exit if no string entered.
  For i = 1 To Len(s$)
    mesg = mesg & Asc(Mid$(s$,i,1)) & crlf
  Next i
  Session.Echo "The Asc values of the string are:" & mesg
End Sub
```

See Also Chr, Chr$, ChrB, ChrB$, ChrW, ChrW$; Character and String Manipulation on page 3

AskBox, AskBox$

Syntax `AskBox[$](prompt$ [,[default$] [,[title$][,helpfile,context]]])`

Description Displays a dialog requesting input from the user and returns that input as a **`String`**. The **`AskBox/ AskBox$`** functions take the following parameters:

Parameter	Description
`prompt$`	String containing the text to be displayed above the text box. The dialog is sized to the appropriate width depending on the width of **`prompt$`**. A runtime error is generated if **`prompt$`** is null.
`default$`	String containing the initial content of the text box. The user can return the default by immediately selecting OK. A runtime error is generated if **`default$`** is null.
`title$`	String specifying the title of the dialog. If missing, then the default title is used.
`helpfile`	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then **`context`** must also be specified.
`Context`	Number specifying the ID of the topic within **`helpfile`** for this dialog's help. If this parameter is specified, then **`helpfile`** must also be specified.

`The AskBox$` function returns a **`String`** containing the input typed by the user in the text box. A zero-length string is returned if the user selects Cancel.

The **`AskBox`** function returns a **`String`** variant containing the input typed by the user in the text box. An **`Empty`** variant is returned if the user selects Cancel.

When the dialog is displayed, the text box has the focus.

The user can type a maximum of 255 characters into the text box displayed by **`AskBox$`**.

If both the **`helpfile`** and **`context`** parameters are specified, then a Help button is added in addition to the OK and Cancel buttons. Context-sensitive help can be invoked by selecting this button or using the help key F1. Invoking help does not remove the dialog.

Example

```
Sub Main
  s$ = AskBox$("Type in the filename:")
  Session.Echo "The filename was: " & s$
End Sub
```

See Also User Interaction on page 16

AskPassword, AskPassword$

Syntax **`AskPassword[$](prompt$ [,[title] [,helpfile,context]])`**

Description Returns a **`String`** containing the text that the user typed. Unlike the **`AskBox/AskBox$`** functions, the user sees asterisks in place of the characters that are actually typed. This allows the hidden input of passwords. The **`AskPassword/AskPassword$`** functions take the following parameters:

Parameter	Description
`prompt$`	String containing the text to be displayed above the text box. The dialog is sized to the appropriate width depending on the width of **`prompt$`**. A runtime error is generated if **`prompt$`** is null.
`title$`	String specifying the title of the dialog. If missing, then the default title is used.
`helpfile`	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then **`context`** must also be specified.
`Context`	Number specifying the ID of the topic within **`helpfile`** for this dialog's help. If this parameter is specified, then **`helpfile`** must also be specified.

When the dialog is first displayed, the text box has the focus.

A maximum of 255 characters can be typed into the text box.

The **`AskPassword$`** function returns the text typed into the text box, up to a maximum of 255 characters. A zero-length string is returned if the user selects Cancel.

The **`AskPassword`** function returns a **`String`** variant. An **`Empty`** variant is returned if the user selects Cancel.

If both the **`helpfile`** and **`context`** parameters are specified, then a Help button is added in addition to the OK and Cancel buttons. Context-sensitive help can be invoked by selecting this button or using the help key F1. Invoking help does not remove the dialog.

Example

```
Sub Main
  s$ = AskPassword$("Type in the password:")
  Session.Echo "The password entered is: " & s$
End Sub
```

See Also User Interaction on page 16

Atn

Syntax **`Atn(number)`**

Description Returns the angle (in radians) whose tangent is **`number`**. Some helpful conversions:

- Pi (3.1415926536) radians = 180 degrees.
- 1 radian = 57.2957795131 degrees.
- 1 degree = .0174532925 radians.

Example

```
Sub Main
  a# = Atn(1.00)
  Session.Echo "1.00 is the tangent of " & a# & " radians (45 degrees)."
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

B

Beep

Syntax **`Beep`**

Description Makes a single system beep.

Example

```
Sub Main
  For i = 1 To 5
    Beep
    Sleep(200)
  Next i
  Session.Echo "You have an upcoming appointment!"
End Sub
```

See Also Operating System Control on page 15

Begin Dialog

Syntax

```
Begin Dialog DialogName [x],[y],width,height,title$ [,[.DlgProc] [,[PicName$]
[,style]]]
    Dialog Statements
End Dialog
```

Description Defines a dialog template for use with the **`Dialog`** statement and function. A dialog template is constructed by placing any of the following statements between the **`Begin Dialog`** and **`End Dialog`** statements (no other statements besides comments can appear within a dialog template).

Note It is easiest to construct a dialog using the dialog editor.

`Picture`	**`PictureButton`**	**`OptionButton`**
`OptionGroup`	**`CancelButton`**	**`Text`**
`TextBox`	**`GroupBox`**	**`DropListBox`**
`ListBox`	**`ComboBox`**	**`CheckBox`**
`PushButton`	**`OKButton`**	

The **`Begin Dialog`** statement requires the following parameters:

Parameter	Description
`x`, **`y`**	Integer coordinates specifying the position of the upper left corner of the dialog relative to the parent window. These coordinates are in dialog units. If either coordinate is unspecified, then the dialog will be centered in that direction on the parent window.
`width`, **`height`**	Integer coordinates specifying the width and height of the dialog (in dialog units).
`DialogName`	Name of the dialog template. Once a dialog template has been created, a variable can be dimensioned using this name.
`title$`	String containing the name to appear in the title bar of the dialog.
`.DlgProc`	Name of the dialog function. The routine specified by **`.DlgProc`** will be called when certain actions occur during processing of the dialog. (See **`DlgProc [prototype]`** for additional information about dialog functions.)If this parameter is omitted, then the compiler processes the dialog using the default dialog processing behavior.
`PicName$`	String specifying the name of a DLL containing pictures. This DLL is used as the origin for pictures when the picture type is 10. If this parameter is omitted, then no picture library will be used.
`style`	Specifies extra styles for the dialog. It can be any of the following values: 0 Dialog does not contain a title or close box. 1 Dialog contains a title and no close box. 2 (or omitted) Dialog contains both the title and close box.

There is an error if the dialog template contains no controls.

A dialog template must have at least one **`PushButton`**, **`OKButton`**, or **`CancelButton`** statement. Otherwise, there will be no way to close the dialog.

Dialog units are defined as 1/4 the width of the font in the horizontal direction and 1/8 the height of the font in the vertical direction.

Any number of user dialoges can be created, but each one must be created using a different name as the **`DialogName`**. Only one user dialog may be invoked at any time.

Expression Evaluation within the dialog Template

The **Begin Dialog** statement creates the template for the dialog. Any expression or variable name that appears within any of the statements in the dialog template is not evaluated until a variable is dimensioned of type **DialogName**. The following example shows this behavior:

```
MyTitle$ = "Hello, World"
Begin Dialog MyTemplate 16,32,116,64,MyTitle$
  OKButton 12,40,40,14
End Dialog
MyTitle$ = "Sample Dialog"
Dim Dummy As MyTemplate
rc% = Dialog(Dummy)
```

The above example creates a dialog with the title "Sample Dialog".

Expressions within dialog templates cannot reference external subroutines or functions.

All controls within a dialog use the same font. The fonts used for the text and text box controls can be changed explicitly by setting the font parameters in the **Text** and **TextBox** statements. A maximum of 128 fonts can be used within a single dialog, although the practical limitation may be less.

Example

```
Sub Main
  Begin Dialog QuitDialogTemplate 16,32,116,64,"Quit"
    Text 4,8,108,8,"Are you sure you want to exit?"
    CheckBox 32,24,63,8,"Save Changes",.SaveChanges
    OKButton 12,40,40,14
    CancelButton 60,40,40,14
  End Dialog
  Dim QuitDialog As QuitDialogTemplate
  rc% = Dialog(QuitDialog)
End Sub
```

See Also User Interaction on page 16

Boolean (data type)

Syntax **Boolean**

Description A data type capable of representing the logical values **True** and **False**. **Boolean** variables are used to hold a binary value—either **True** or **False**. There is no type-declaration character for **Boolean** variables. Variables can be declared as **Boolean** using the **Dim**, **Public**, or **Private** statement. Internally, a **Boolean** variable is a 2-byte value holding –1 (for **True**) or 0 (for **False**). When appearing as a structure member, **Boolean** members require 2 bytes of storage; When used within binary or random files, 2 bytes of storage are required.

Any type of data can be assigned to **Boolean** variables. **Boolean** variables that have not yet been assigned are given an initial value of **False**.When assigning, non-0 values are converted to **True**, and 0 values are converted to **False**. Variants can hold **Boolean** values when assigned the results of comparisons or the constants **True** or **False**. When passed to external routines, **Boolean** values are sign-extended to the size of an integer on that platform (either 16 or 32 bits) before pushing onto the stack.

See Also Keywords, Data Types, Operators, and Expressions on page 6

ByRef

Syntax `...,ByRef parameter,...`

Description Used within the **`Sub`**...**`End Sub`**, **`Function`**...**`End Function`**, or **`Declare`** statement to specify that a given parameter can be modified by the called routine.

Note Passing a parameter by reference means that the caller can modify that variable's value.

Unlike the **`ByVal`** keyword, the **`ByRef`** keyword cannot be used when passing a parameter. The absence of the **`ByVal`** keyword is sufficient to force a parameter to be passed by reference:

```
MySub ByVal i    'Pass i by value.
MySub ByRef i    'Illegal (will not compile).
MySub i      'Pass i by reference.
```

Example

```
Sub Test(ByRef a As Variant)
  a = 14
End Sub

Sub Main
  b = 12
  Test b
  Session.Echo "The ByRef value is: " & b      'Displays 14.
End Sub
```

See Also ( ) (precedence), ByVal; Keywords, Data Types, Operators, and Expressions on page 6; Macro Control and Compilation on page 10

ByVal

Syntax `...ByVal parameter...`

Description Forces a parameter to be passed by value rather than by reference. The **`ByVal`** keyword can appear before any parameter passed to any function, statement, or method to force that parameter to be passed by value. Passing a parameter by value means that the caller cannot modify that variable's value. Enclosing a variable within parentheses has the same effect as the **`ByVal`** keyword:

```
Foo ByVal   i    'Forces i to be passed by value.
Foo(i)        'Forces i to be passed by value.
```

When calling external statements and functions (i.e., routines defined using the **`Declare`** statement), the **`ByVal`** keyword forces the parameter to be passed by value regardless of the declaration of that parameter in the **Declare** statement. The following example shows the effect of the **ByVal** keyword used to passed an **Integer** to an external routine:

```
Declare Sub Foo Lib "MyLib" (ByRef i As Integer)
i% = 6
Foo ByVal i%    'Pass a 2-byte Integer.
Foo i%      'Pass a 4-byte pointer to an Integer.
```

Since the **Foo** routine expects to receive a pointer to an **Integer**, the first call to **Foo** will have unpredictable results.

Example

```
Sub Foo(a As Integer)
  a = a + 1
End Sub

Sub Main
  Dim i As Integer
  i = 10
  Foo i
  Session.Echo "The ByVal value is: " & i    'Displays 11
                '(Foo changed the value).
  Foo ByVal i
  Session.Echo "The ByVal value is still: " & i  'Displays 11 Foo did not _
change the value).
End Sub
```

See Also ( ) (precedence), ByRef; Keywords, Data Types, Operators, and Expressions on page 6; Macro Control and Compilation on page 10

C

Call

Syntax `Call subroutine_name [(arguments)]`

Description Transfers control to the given subroutine, optionally passing the specified arguments. Using this statement is equivalent to:

```
subroutine_name [arguments]
```

Use of the **`Call`** statement is optional. The **`Call`** statement can only be used to execute subroutines; functions cannot be executed with this statement. The subroutine to which control is transferred by the **`Call`** statement must be declared outside of the calling procedure, as shown in the following example.

Examples This example uses the Call statement to pass control to another function.

```
Sub Example_Call(s$)
  'This subroutine is declared externally to Main and displays
  'the text passed in the parameter s$.
  Session.Echo "Call: " & s$
End Sub

Sub Main
'This example assigns a string variable to display, then calls
'subroutine Example_Call, passing parameter s$ to be displayed within
'the subroutine.

  s$ = "DAVE"
  Example_Call s$
  Call Example_Call("SUSAN")
End Sub
```

See Also Macro Control and Compilation on page 10

CancelButton

Syntax **`CancelButton x, y, width, height [,.Identifier]`**

Description Defines a Cancel button that appears within a dialog template. This statement can only appear within a dialog template (i.e., between the **`Begin Dialog`** and **`End Dialog`** statements).

Selecting the Cancel button (or pressing Esc) dismisses the user dialog, causing the **`Dialog`** function to return 0. (Note: A dialog function can redefine this behavior.) Pressing the Esc key or double-clicking the close box will have no effect if a dialog does not contain a **`CancelButton`** statement.

The **`CancelButton`** statement requires the following parameters:

Parameter	Description
`x`, **`y`**	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
`width`, **`height`**	Integer coordinates specifying the dimensions of the control in dialog units.
`.Identifier`	Optional parameter specifying the name by which this control can be referenced by statements in a dialog function (such as **`DlgFocus`** and **`DlgEnable`**). If this parameter is omitted, then the word **`"Cancel"`** is used.

A dialog must contain at least one **`OKButton`**, **`CancelButton`**, or **`PushButton`** statement; otherwise, the dialog cannot be dismissed.

Example

```
Sub Main
  Begin Dialog SampleDialogTemplate 37,32,48,52,"Sample"
    OKButton 4,12,40,14,.OK
    CancelButton 4,32,40,14,.Cancel
  End Dialog
  Dim SampleDialog As SampleDialogTemplate
  r% = Dialog(SampleDialog)
  If r% = 0 Then Session.Echo "Cancel was pressed!"
End Sub
```

See Also User Interaction on page 16

CBool

Syntax **`CBool(expression)`**

Description Converts **`expression`** to **`True`** or **`False`**, returning a **`Boolean`** value. The **`expression`** parameter is any expression that can be converted to a **`Boolean`**. A runtime error is generated if **`expression`** is **`Null`**.

All numeric data types are convertible to **`Boolean`**. If **`expression`** is zero, then the **`CBool`** returns **`False`**; otherwise, **`CBool`** returns **`True`**. **`Empty`** is treated as **`False`**.

If **expression** is a **String**, then **CBool** first attempts to convert it to a number, then converts the number to a **Boolean**. A runtime error is generated if **expression** cannot be converted to a number.

A runtime error is generated if **expression** cannot be converted to a **Boolean**.

Example

```
Sub Main
  Dim IsNumericOrDate As Boolean
  s$ = "34224.54"
  IsNumericOrDate = CBool(IsNumeric(s$) Or IsDate(s$))
  If IsNumericOrDate = True Then
    Session.Echo s$ & " is either a valid date or number!"
  Else
    Session.Echo s$ & " is not a valid date or number!"
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

CCur

Syntax `CCur(expression)`

Description Converts any expression to a **Currency**. This function accepts any expression convertible to a **Currency**, including strings. A runtime error is generated if **expression** is **Null** or a **String** not convertible to a number. **Empty** is treated as 0.

When passed a numeric expression, this function has the same effect as assigning the numeric expression number to a **Currency**.

When used with variants, this function guarantees that the variant will be assigned a **Currency** (**VarType** 6).

Example

```
Sub Main
  i$ = "100.44"
  Session.Echo "The currency value is: " & CCur(i$)
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

CDate, CVDate

Syntax **`CDate(expression)`**
`CVDate(expression)`

Description Converts **`expression`** to a date, returning a **`Date`** value. The **`expression`** parameter is any expression that can be converted to a **`Date`**. A runtime error is generated if **`expression`** is **`Null`**.

If **`expression`** is a **`String`**, an attempt is made to convert it to a **`Date`** using the current country settings. If **`expression`** does not represent a valid date, then an attempt is made to convert **`expression`** to a number. A runtime error is generated if **`expression`** cannot be represented as a date.

These functions are sensitive to the date and time formats of your computer.

Note The **`CDate`** and **`CVDate`** functions are identical.

Example

```
Sub Main
  Dim date1 As Date
  Dim date2 As Date
  Dim diff As Date
  date1 = CDate(#1/1/1994#)
  date2 = CDate("February 1, 1994")
  diff = DateDiff("d",date1,date2)
  Session.Echo "The date difference is " & CInt(diff) & " days."
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Time and Date Access on page 17

CDbl

Syntax **`CDbl(expression)`**

Description Converts any expression to a **`Double`**. This function accepts any expression convertible to a **`Double`**, including strings. A runtime error is generated if **`expression`** is **`Null`**. **`Empty`** is treated as 0.0.

When passed a numeric expression, this function has the same effect as assigning the numeric expression number to a **`Double`**.

When used with variants, this function guarantees that the variant will be assigned a **`Double`** (**`VarType`** 5).

Example

```
Sub Main
  i% = 100
  j! = 123.44
  Session.Echo "The double value is: " & CDbl(i% * j!)
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

ChDir

Syntax `ChDir path`

Description Changes the current directory of the specified drive to **`path`**. This routine will not change the current drive. (See **`ChDrive`** [**`statement`**].)

Example

```
Const crlf = $(13) + Chr$(10)

Sub Main
  save$ = CurDir$
  ChDir ("C:\")
  Session.Echo "Old: " & save$ & crlf & "New: " & CurDir$
  ChDir (save$)
  Session.Echo "Directory restored to: " & CurDir$
End Sub
```

See Also Drive, Folder, and File Access on page 4

ChDrive

Syntax `ChDrive drive`

Description Changes the default drive to the specified drive. Only the first character of **`drive`** is used. Also, **`drive`** is not case-sensitive. If **`drive`** is empty, then the current drive is not changed.

Example

```
Const crlf$ = Chr$(13) + Chr$(10)

Sub Main
  cd$ = CurDir$
  save$ = Mid$(CurDir$,1,1)
  If save$ = "D" Then
    ChDrive("C")
  Else
    ChDrive("D")
  End If
  Session.Echo "Old: " & save$ & crlf & "New: " & CurDir$
  ChDrive (save$)
  Session.Echo "Directory restored to: " & CurDir$
End Sub
```

See Also Drive, Folder, and File Access on page 4

CheckBox

Syntax `CheckBox x, y, width, height, title$, .Identifier`

Description Defines a checkbox within a dialog template. Checkbox controls are either on or off, depending on the value of **`.Identifier`**. This statement can only appear within a dialog template (i.e., between the **`Begin Dialog`** and **`End Dialog`** statements). The **`CheckBox`** statement requires the following parameters:

Parameter	Description
x, **y**	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
width, **height**	Integer coordinates specifying the dimensions of the control (in dialog units).
title$	String containing the text that appears within the checkbox. This text may contain an ampersand character to denote an accelerator letter, such as "&Font" for Font (indicating that the Font control may be selected by pressing the F accelerator key).
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). This parameter also creates an integer variable whose value corresponds to the state of the checkbox (1 = checked; 0 = unchecked). This variable can be accessed using the syntax: **DialogVariable.Identifier**.

When the dialog is first created, the value referenced by **.Identifier** is used to set the initial state of the checkbox. When the dialog is dismissed, the final state of the checkbox is placed into this variable. By default, the **.Identifier** variable contains 0, meaning that the checkbox is unchecked.

Accelerators are underlined, and the accelerator combination Alt+**letter** is used.

Example

```
Sub Main
  Begin Dialog SaveOptionsTemplate 36,32,151,52,"Save"
    GroupBox 4,4,84,40,"GroupBox"
    CheckBox 12,16,67,8,"Include heading",.IncludeHeading
    CheckBox 12,28,73,8,"Expand keywords",.ExpandKeywords
    OKButton 104,8,40,14,.OK
    CancelButton 104,28,40,14,.Cancel
  End Dialog
  Dim SaveOptions As SaveOptionsTemplate
  SaveOptions.IncludeHeading = 1    'Checkbox initially on.
  SaveOptions.ExpandKeywords = 0    'Checkbox initially off.
  r% = Dialog(SaveOptions)
  If r% = -1 Then
    Session.Echo "OK was pressed."
  End If
End Sub
```

See Also User Interaction on page 16

Choose

Syntax **Choose(index,expression1,expression2,...,expression13)**

Description Returns the expression at the specified index position. The **index** parameter specifies which expression is to be returned. If **index** is 1, then **expression1** is returned; if **index** is 2, then **expression2** is returned, and so on. If **index** is less than 1 or greater than the number of supplied expressions, then **Null** is returned.

The **index** parameter is rounded down to the nearest whole number.

The **Choose** function returns the expression without converting its type. Each expression is evaluated before returning the selected one.

Example

```
Sub Main
  Dim a As Variant
  Dim c As Integer
  c% = 2
  a = Choose(c%,"Hello, world",#1/1/94#,5.5,False)
  'Display the date passed as a parameter:
  Session.Echo "Item " & c% & " is '" & a & "'"
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Chr, Chr$, ChrB, ChrB$, ChrW, ChrW$

Syntax

```
Chr[$](charcode)
ChrB[$](charcode)
ChrW[$](charcode)
```

Description Returns the character the value of which is **charcoode**. The **Chr$**, **ChrB$**, and **ChrW$** functions return a **String**, whereas the **Chr**, **ChrB**, and **ChrW** functions return a **String** variant. These functions behave differently depending on the string format:

Function	String Format	Value between	Returns
Chr[$]	SBCS	0 and 255	1-byte character string.
	MBCS	-32768 and 32767	1-byte or 2-byte MBCS character string depending on **charcode**.
	Wide	-32768 and 32767	2-byte character string.
ChrB[$]	SBCS	0 and 255	1-byte character string.
	MBCS	0 and 255	1-byte character string.
	Wide	0 and 255	2-byte character string.
ChrW[$]	SBCS	0 and 255	1-byte character string (same as **Chr** and **Chr$** functions)
	MBCS	-32768 and 32767	1-byte or 2-byte MBCS character string depending on **charcode**.
	Wide	-32768 and 32767	2-byte character string.

The **Chr$** function can be used within constant declarations, as in the following example:

```
Const crlf = Chr$(13) + Chr$(10)
```

Some common uses of this function are:

Function	Use
`Chr$(9)`	Tab
`Chr$(13) + Chr$(10)`	End-of-line (carriage return, linefeed)
`Chr$(26)`	End-of-file
`Chr$(0)`	Null

Examples Concatenates carriage return (13) and line feed (10) in **`crlf$`**, then displays a multiple-line message using **`crlf$`** to separate lines.

```
Sub Main
  crlf$ = Chr$(13) + Chr$(10)
  Session.Echo "First line." & crlf$ & "Second line."
  'Fills an array with the ASCII characters for ABC and
  'displays their corresponding characters.
  Dim a%(2)
  For i = 0 To 2
    a%(i) = (65 + i)
  Next I
  Session.Echo "The first three elements of the array are: " & Chr$(a%(0)) &
Chr$(a%(1)) & Chr$(a%(2))
End Sub
```

See Also Character and String Manipulation on page 3

CInt

Syntax **`CInt(expression)`**

Description Converts **`expression`** to an **`Integer`**. This function accepts any expression convertible to an **`Integer`**, including strings. A runtime error is generated if **`expression`** is **`Null`**. **`Empty`** is treated as 0. The passed numeric expression must be within the valid range for integers:

```
-32768 <= expression <= 32767
```

A runtime error results if the passed expression is not within the above range.

When passed a numeric expression, this function has the same effect as assigning a numeric expression to an **`Integer`**. Note that integer variables are rounded before conversion.

When used with variants, this function guarantees that the expression is converted to an **`Integer`** variant (**`VarType`** 2).

Example

```
Sub Main
  '(1) Assigns i# to 100.55 and displays its integer representation (101).
  i# = 100.55
  Session.Echo "The value of CInt(i) = " & CInt(i#)
  '(2) Sets j# to 100.22 and displays the CInt
  'representation (100).
  j# = 100.22
```

```
  Session.Echo "The value of CInt(j) = " & CInt(j#)
  '(3) Assigns k% (integer) to the CInt sum of j# and k% and
  'displays k% (201).
  k% = CInt(i# + j#)
   Session.Echo "The integer sum of 100.55 and 100.22 is: " & k%
  '(4) Reassigns i# to 50.35 and recalculates k%, then
  'displays the result (note rounding).
  i# = 50.35
  k% = CInt(i# + j#)
  Session.Echo "The integer sum of 50.35 and 100.22 is: " & k%
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Circuit (object)

Circuit methods and properties indicate the scope of their action by their name by incorporating the appropriate communication method in the name (such as Circuit.LATHostName). Properties and methods common to all communication methods do not incorporate a communication method name (such as Circuit.AssertBreak). As of this version of SmarTerm, the supported communication methods are LAT, modem, serial, SNA, and Telnet.

Circuit.AssertBreak

Syntax `Circuit.AssertBreak`

Description Asserts a communications break and returns a boolean representing the completion status. This method asserts a communications `Break` condition appropriate for the communications method being used.

Example

```
Sub Main
  Dim BreakStatus as Boolean
  BreakStatus = Circuit.AssertBreak()
  If BreakStatus = FALSE Then
     Session.Echo "An error occurred"
  End If
End Sub
```

See Also Host Connections on page 7; Objects on page 18

Circuit.AutoConnect

Syntax `Circuit.AutoConnect`

Description Returns or sets the communication method's autoconnect state (boolean).

Example

```
Sub Main
  Dim StAuto as Boolean
  StAuto = Circuit.AutoConnect
  If StAuto = False Then
     Session.Echo "Turning autoconnect on"
```

```
    Circuit.AutoConnect = True
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.Connect

Syntax `Circuit.Connect`

Description Establishes a connection to a host and always returns a value of True. Use Circuit.Connected if you want to check connection status.

Example

```
Sub Main
  If Circuit.Connected Then
    If Circuit.Disconnect = FALSE Then
      Session.Echo "Disconnect error"
    End If
  End If
  Circuit.TelnetPortNumber = 21
  Circuit.TelnetHostName = "SomeHost.com"
  If Circuit.Connect = FALSE Then
    Session.Echo "Connect error"
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.Connected

Syntax `Circuit.Connected`

Description Returns a boolean representing the session's connection state.

Example

```
Sub Main
  If Circuit.Connected Then
    Circuit.Disconnect
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.Disconnect

Syntax `Circuit.Disconnect`

Description Disconnects from the host and returns a boolean representing the completion status.

Example

```
Sub Main
  If Circuit.Connected Then
    Circuit.Disconnect
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.LATHostName

Syntax `Circuit.LATHostName`

Description Returns or sets the host name for the LAT communications driver (string).

Example

```
Sub Main
  Dim HostName as String
  HostName = Circuit.LATHostName
  If HostName <> "LATHost1" Then
     Session.Echo "Setting the host to LATHost1 to read your email"
     Circuit.LATHostName =  "LATHost1"
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.LATPassword

Syntax `Circuit.LATPassword`

Description Returns or sets the password for the LAT communications driver (string).

Example

```
Sub Main
  Dim Password, NewPass as String
  Password = Circuit.LATPassword
  If Password = "" Then
     NewPass = AskPassword$("Type in your LAT password.")
     Circuit.LATPassword = NewPass
End Sub
```

See Also Host Connections on page 7

Circuit.LATSavePassword

Syntax `Circuit.LATSavePassword`

Description Returns or sets if a password will be saved for the LAT communications driver.

Example

```
Sub Main
  Dim SavePassState as Boolean
  SavePassState = Circuit.LATSavePassword
  If SavePassState = True Then
     Session.Echo "For security reasons, you cannot save your password"
     Circuit.LATSavePassword = False
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.ModemAlt1Number

Syntax `Circuit.ModemAlt1Number`

Description Returns or sets the first alternate phone number to be used when making a modem connection (string).

Example

```
Sub Main
  Dim PhoneNumberAlt1 as String
  PhoneNumberAlt1 = Circuit.ModemAlt1Number
  If PhoneNumberAlt1 = "" Then
     Circuit.ModemAlt1Number = "555-1234"
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.ModemAlt2Number

Syntax `Circuit.ModemAlt2Number`

Description Returns or sets the second alternate phone number to be used when making a modem connection (string).

Example

```
Sub Main
  Dim PhoneNumberAlt2 as String
  PhoneNumberAlt2 = Circuit.ModemAlt2Number
  If PhoneNumberAlt2 = "" Then
     Circuit.ModemAlt2Number = "555-1212"
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.ModemAlt3Number

Syntax `Circuit.ModemAlt3Number`

Description Returns or sets the third alternate phone number to be used when making a modem connection (string).

Example

```
Sub Main
  Dim PhoneNumberAlt3 as String
  PhoneNumberAlt3 = Circuit.ModemAlt3Number
  If PhoneNumberAlt3 = "" Then
     Circuit.ModemAlt3Number = "555-1212"
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.ModemAreaCode

Syntax `Circuit.ModemAreaCode`

Description Returns or sets the area code to be used when making a modem connection (string).

Example

```
Sub Main
  Dim AreaCode as String
  AreaCode = Circuit.ModemAreaCode
  If AreaCode = "" Then
     Circuit.ModemAreaCode = "800"
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.ModemCountryCode

Syntax `Circuit.ModemCountryCode`

Description Returns or sets the current country code.

Example See example for Circuit.ModemGetCountryCodeString.

See Also Host Connections on page 7

Circuit.ModemGetCountryCodeString

Syntax `Circuit.ModemGetCountryCodeString index`

where **`index`** is a 1-based index into the set of country code strings.

Description Returns a string representing the indexed country code.

Example

```
Option base 1
Sub Main
    Dim TotalStrings as Integer
    Dim CountryCodes(TotalStrings) as String
    Dim i as Integer
   'Fill the CountryCodes array
    TotalStrings = Circuit.ModemTotalCountryCodes
    For i = 1 to TotalStrings
        CountryCodes(i) = Circuit.ModemGetCountryCodeString(i)
    Next i
    Session.Echo "Current country code: " & Circuit.ModemCountryCode
   'Choose a new country code
    Circuit.ModemCountryCode = CountryCodes(4)
    Session.Echo "New country code: " & Circuit.ModemCountryCode
End Sub
```

See Also Host Connections on page 7

Circuit.ModemPhoneNumber

Syntax `Circuit.ModemPhoneNumber`

Description Returns or sets the primary phone number to be used when making a modem connection (string).

Example

```
Sub Main
  Dim PhoneNumber as String
  PhoneNumber = Circuit.ModemPhoneNumber
  Session.Echo "The current phone number is " & PhoneNumber
  Circuit.ModemPhoneNumber = "555-1212"
End Sub
```

See Also Host Connections on page 7

Circuit.ModemTotalCountryCodes

Syntax `Circuit.ModemTotalCountryCodes`

Description Returns an integer representing the total number of country code strings available through the `Circuit.ModemGetCountryCodeString` method.

Example See example for Circuit.ModemGetCountryCodeString.

See Also Host Connections on page 7

Circuit.ModemUseCodes

Syntax `Circuit.ModemUseCodes`

Description Returns or sets whether or not the country code and area code values should be used when dialing (boolean).

Example

```
Sub Main
  Dim CurrentUseCodes as Boolean
  CurrentUseCodes = Circuit.ModemUseCodes
    If CurrentUseCodes = FALSE Then
     Session.Echo "The country code and area code will be used"
Circuit.ModemUseCodes = True
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.SendRawToHost

Syntax `Circuit.SendRawToHost (data, datalength)`

Description Sends data to host without character translation and without 8 bit to 7 bit control mapping. Returns the operation’s completion status (boolean). Parameters are:

Parameter	Description
`data`	Variant, the data to send.
`Datalength`	Integer, size of the data (in bytes)

Example

```
Sub Main
  Dim fSuccess as Boolean
  fSuccess = Circuit.SendRawToHost("12345", 5)
  If fSuccess = FALSE Then
     Session.Echo "An error occurred."
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.SerialBaudRate

Syntax `Circuit.SerialBaudRate`

Description Returns or sets the serial driver's current baud rate (long integer)

`Circuit.SerialBaudRate` accepts or returns one of the following values: **`1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600,`** or **`115200.`**

Example

```
Sub Main
  Dim BaudRate as Long
   BaudRate = Circuit.SerialBaudRate
  If BaudRate < 9600 Then
     Session.Echo "This connection needs a baud rate of at least 9600 baud"
     Circuit.SerialBaudRate = 9600
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.SerialBreakDuration

Syntax `Circuit.SerialBreakDuration`

Description Returns or sets an integer containing the serial driver's current break duration value (integer). **`Circuit.SerialBreakDuration`** accepts or returns one of the following values:

Value	Definition
375	Break duration of 375ms
2000	Break duration of 2000ms

Example

```
Sub Main
   Dim BreakTime as Integer
  BreakTime = Circuit.SerialBreakDuration
  Circuit.SerialBreakDuration = 375
End Sub
```

See Also Host Connections on page 7

Circuit.SerialDataBits

Syntax `Circuit.SerialDataBits`

Description Returns or sets the serial driver's current data bits value (integer). `Circuit.SerialDataBits` accepts or returns one of the following values:

Value	Definition
7	Configure for 7 data bits.
8	Configure for 8 data bits.

Example

```
Sub Main
  Dim DataBits as Integer
  DataBits = Circuit.SerialDataBits
  If DataBits = 7 Then
     Session.Echo "This connection requires an 8-bit connection"
     Circuit.SerialDataBits = 8
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.SerialFlowControl

Syntax `Circuit.SerialFlowControl`

Description Returns or sets the serial driver's current flow control setting (integer). Possible values are:

Value	Constant	Meaning
0	`smlNOFLOWCONTROL`	No flow control.
1	`smlXONXOFF`	XON/XOFF flow control.
2	`smlRTSCTS`	RTS/CTS flow control.
3	`smlDTRDSR`	DTR/DSR flow control.

Example

```
Sub Main
  Dim FlowControl as Integer
  FlowControl = Circuit.SerialFlowControl
  If FlowControl = smlRTSCTS Then
     Circuit.SerialFlowControl = smlXONXOFF
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.SerialParity

Syntax `Circuit.SerialParity`

Description Returns or sets the serial driver's current parity setting (integer). Possible values are:

Value	Constant	Meaning
0	`smlNOPARITY`	No parity.
1	`smlODDPARITY`	Odd parity.
2	`smlEVENPARITY`	Even parity.
3	`smlMARKPARITY`	Mark parity.
4	`smlSPACEPARITY`	Space parity.

Example

```
Sub Main
  Dim Parity as Integer
  Parity = Circuit.SerialParity
  Circuit.SerialParity = smlODDPARITY
End Sub
```

See Also Host Connections on page 7

Circuit.SerialPort

Syntax `Circuit.SerialPort`

Description Returns or sets the serial driver's current port number (integer). `Circuit.SerialPort` accepts or returns a value within the range: 1 - 255.

Example

```
Sub Main
  Dim ComPort as Integer
  ComPort = Circuit.SerialPort
  If ComPort > 2 Then
     Session.Echo "Setting communications port to COM1"
     Circuit.SerialPort = 1
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.SerialReceiveBufferSize

Syntax `Circuit.SerialReceiveBufferSize`

Description Returns or sets the serial driver's current receive buffer size (integer). Accepts or returns one of the following values: **512, 1024, 2048, 4096,** or **8192**.

Example

```
Sub Main
  Dim ReceiveBufferSize as Integer
  ReceiveBufferSize = Circuit.SerialReceiveBufferSize
  If ReceiveBufferSize < 8192 Then
     Session.Echo "Changing your Buffer size to 8192"
     Circuit.SerialReceiveBufferSize = 8192
  End If
End Sub
```

See Also `Circuit.Connect` (method)

Circuit.SerialStopBits

Syntax **`Circuit.SerialStopBits`**

Description Returns or sets the serial driver's current stop bits value (integer). This property accepts or returns one of the following values:

Value	Definition
1	1 stop bit
2	2 stop bits

Example

```
Sub Main
  Dim StopBits as Integer
  StopBits = Circuit.SerialStopBits
  If StopBits <> 1 Then
     Session.Echo "This connection requires 1 stop bit"
     Circuit.SerialStopBits = 1
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.SerialTransmitBufferSize

Syntax **`Circuit.SerialTransmitBufferSize`**

Description Returns or sets the serial driver's current transmit buffer size (integer). `Circuit.SerialTransmitBufferSize` accepts or returns one of the following values: **`512, 1024, 2048, 4096,`** or **`8192.`**

Example

```
Sub Main
  Dim TransmitBufferSize as Integer
  TransmitBufferSize = Circuit.SerialTransmitBufferSize
  If TransmitBufferSize < 8192 Then
     Session.Echo "Changing your Buffer size to 8192"
     Circuit.SerialTransmitBufferSize = 8192
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.Setup

Syntax **`Circuit.Setup setupstring$`**

where **`setupstring$`** is the string containing the setup specifications (string).

Description Sets SmarTerm communications parameters. This method is provided primarily for the support of PSL scripts.

The syntax of the string expression is identical between communication methods, although meaning varies somewhat. Specify setup options one at a time with their own **`Circuit.Setup`** statements, or more than one at a time, if you keep all options and settings within the quotation marks, separating the setup statements with commas:

```
Circuit.Setup  "baudrate = 2400, parity = NONE, stopbits = 1"
```

Serial COM1-COM4

```
Serial Port
portname= COM1 | COM2 | COM3 | COM4
Circuit.Setup "portname = COM1"
Baud Rate
baudrate= 1200 | 2400 | 4800 | 9600 | 19200 | 38400 | 57600
Circuit.Setup "baudrate = 2400"
Data Bits
bytesize= 7 | 8
Circuit.Setup "bytesize = 7"
Stop Bits
stopbits= 1 | 2
Circuit.Setup "stopbits = 1"
Parity
parity= NONE | ODD | EVEN | MARK | SPACE
Circuit.Setup "parity = even"
Break Duration
breaktime= 375 | 2000
Circuit.Setup "breaktime = 2000"
Flow Control
flowcontrol= XON/XOFF | RTS/CTS | DTR/DSR | NONE
Circuit.Setup "flowcontrol = dtr/dsr"
Receive Buffer Size
receivequeuesize= 512 | 1024 | 2048 | 4096 | 8196
Circuit.Setup "receivequeuesize = 512"
Transmit Buffer Size
transmitqueuesize= 512 | 1024 | 2048 | 4096 | 8196
Circuit.Setup "transmitqueuesize = 512"
Autoconnect on configuration open
autoconnect= TRUE | FALSE
Circuit.Setup "autoconnect = true"
```

Telnet

```
Host name or IP Address
hostname= ASCII string of no more than 60 characters
Circuit.Setup "hostname = unixbox"
Port Number
portnumber= Decimal number between 1 and 32767 inclusive
Circuit.Setup "portnumber = 391"
Break Mode
breakmode= INTERRUPT | BREAK
Circuit.Setup "breakmode = interrupt"
Character Mode
charmode= ASCII | BINARY
Circuit.Setup "charmode = ascii"
Auto-connect on configuration open
```

```
autoconnect= TRUE | FALSE
Circuit.Setup "autoconnect = true"
```

See Also Host Connections on page 7

Circuit.SNALogicalUnit

3270 sessions only

Syntax `Circuit.SNALogicalUnit`

Description Returns or sets the LU (logical unit) to which the SmarTerm session connects. Triggers an application-based menu action in SmarTerm. The LU is the access point into the SNA network, allowing SmarTerm to reach a particular host service (for example, a mainframe application LU). The pool name is a name you assign to a set of LUs with the same capabilities. When the session connects, it is automatically given the first available LU in the pool.

Example

```
Sub Main
  Circuit.SNALogicalUnit "LU2"
End Sub
```

See Also Host Connections on page 7

Circuit.SNAProtocol

3270 sessions only

Syntax `Circuit.SNAProtocol`

Description Returns or sets the transfer protocol for the SmarTerm session. Possible values are:

Value	Description
IPX/SPX	Internetwork Packet Exchange/Sequenced Packet Exchange. Novell's protocol used by Novell NetWare. A router with IPX routing can interconnect local area networks so that Novell NetWare clients and servers can communicate.
TCP/IP	Transmission Control Protocol over Internet Protocol. The most common transport layer protocol used on Ethernet and the Internet. This property is supported in NetWare for SAA connections only.

Example

```
'This example
Sub Main
   Circuit.SNAProtocol "TCP/IP"
End Sub
```

See Also Host Connections on page 7

Circuit.SNAServerName

3270 and 5250 sessions only

Syntax `Circuit.SNAServerName`

Description NetWare for SAA connections only.

Returns or sets the name of the server to which the session connects.

Example

```
'This example
Sub Main
   Circuit.SNAServerName " "
End Sub
```

See Also Host Connections on page 7

Circuit.SuppressConnectErrorDialog

Syntax `Circuit.SuppressConnectErrorDialog`

Description Returns or sets the display of connection error dialogs (boolean). If TRUE (the default), then no connection error dialogs are displayed. If FALSE, then all connection error dialogs are displayed.

Common to all communications methods.

Example

```
'This example attempts to connect to one of two hosts.
'using Telnet. If the macro cannot connect to one host,
'it attempts toconnect to the other without informing
'the user of the error

Sub Main

Dim fConnected As Boolean
fConnected = FALSE

'First, turn off connection error dialogs.
Circuit.SuppressConnectErrorDialog = TRUE

'Now, try to connect to the first host
Circuit.TelnetHostName = "MyHost1"
Circuit.Connect

'Give the host 5 seconds to connect. If it connects,
'then go to the next block.
For Seconds = 1 to 5'
   Sleep (1000)
   If Circuit.Connected = TRUE then
      fConnected = TRUE
      Exit For
   End If
Next Seconds

'Now, turn connection error dialogs back on
Circuit.SuppressConnectErrorDialog = FALSE
```

```
'Now determine if we connected to the first host.
'If not, try connecting to the second.
If fConnected = FALSE Then
   Circuit.TelnetHostName = "MyHost2"
   Circuit.Connect
End If

End Sub
```

See Also Host Connections on page 7

Circuit.TelnetBreakMode

Syntax `Circuit.TelnetBreakMode`

Description Returns or sets the Telnet driver's current break mode setting (integer). Possible values are:

Value	Constant	Meaning
0	`smlBREAK`	Set the break mode to break.
1	`smlINTERRUPT`	Set the break mode to interrupt.

Example

```
Sub Main
  Dim BrkMode as Integer
  BrkMode = Circuit.TelnetBreakMode
  If BrkMode = smlBREAK Then
     Session.Echo "Using Interrupt break mode for this connection"
     Circuit.TelnetBreakMode = smlINTERRUPT
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.TelnetCharacterMode

Syntax `Circuit.TelnetCharacterMode`

Description Returns or sets the Telnet driver's current character mode setting (integer). Possible values are:

Value	Constant	Meaning
0	`smlASCII`	Set the character mode to ASCII.
1	`smlBINARY`	Set the character mode to binary.

Example

```
Sub Main
  Dim CharMode as Integer
  CharMode = Circuit.TelnetCharacterMode
  If CharMode = smlASCII Then
     Session.Echo "Changing character mode setting to Binary"
     Circuit.TelnetCharacterMode = smlBinary
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.TelnetHostName

Syntax `Circuit.TelnetHostName`

Description Returns or sets the Telnet driver's current host name (string).

Example

```
Sub Main
  Dim HostName as String
  HostName = Circuit.TelnetHostName
  If HostName = "BrokenHost.com" Then
     Session.Echo "BrokenHost is currently down.  Try WorkingHost.com"
Circuit.TelnetHostName = "WorkingHost.com"
  End If
End Sub
```

See Also Host Connections on page 7

Circuit.TelnetPortNumber

Syntax `Circuit.TelnetPortNumber`

Description Returns or sets the Telnet driver's current port number (string).

Example

```
Sub Main
  Dim Port as String
  Port = Circuit.TelnetPortNumber
  If Port <> 23 Then
     Session.Echo "Setting the port to 23 for a Telnet connection"
     Circuit.TelnetPortNumber = 23
  End If
End Sub
```

See Also Host Connections on page 7

Clipboard (object)

Clipboard$ (function)

Syntax `Clipboard$[()]`

Description Returns a `String` containing the contents of the Clipboard. If the Clipboard doesn't contain text or the Clipboard is empty, then a zero-length string is returned.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Session.Echo "The text in the Clipboard is:" & crlf & Clipboard$
  Clipboard.Clear
  Session.Echo "The text in the Clipboard is:" & crlf & Clipboard$
End Sub
```

See Also Clipboard$ (statement); Operating System Control on page 15

Clipboard$ (statement)

Syntax **`Clipboard$ NewContent$`**

Description Copies **`NewContent$`** into the Clipboard.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Clipboard$ "Hello out there!"
  Session.Echo "The text in the Clipboard is:" & crlf & Clipboard$
  Clipboard.Clear
  Session.Echo "The text in the Clipboard is:" & crlf & Clipboard$
End Sub
```

See Also Clipboard$ (function); Operating System Control on page 15

Clipboard.Clear

Syntax **`Clipboard.Clear`**

Description Clears the Clipboard by removing any content.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Clipboard$ "Hello out there!"
  Session.Echo "The text in the Clipboard is:" & crlf & Clipboard$
  Clipboard.Clear
  Session.Echo "The text in the Clipboard is:" & crlf & Clipboard$
End Sub
```

See Also Clipboard$ (function); Operating System Control on page 15

Clipboard.GetFormat

Syntax **`WhichFormat = Clipboard.GetFormat(format)`**

Description Returns **`True`** if data of the specified format is available in the Clipboard; returns **`False`** otherwise. This method is used to determine whether the data in the Clipboard is of a particular format. The format parameter is an **`Integer`** representing the format to be queried:

Format	Value	Description
`ebCFText`	1	Text
`ebCFBitmap`	2	Bitmap
`ebCFMetafile`	3	Metafile
`ebCFDIB`	8	Device-independent bitmap (DIB)
`ebCFPalette`	9	Color palette
`ebCFUnicodeText`	13	Unicode text

Example

```
Sub Main
  Clipboard$ "Hello out there!"
  If Clipboard.GetFormat(ebCFText) Then
     Session.Echo Clipboard$
  Else
    Session.Echo "There is no text in the Clipboard."
  End If
End Sub
```

See Also Clipboard$ (function); Operating System Control on page 15

Clipboard.GetText

Syntax **`text$ = Clipboard.GetText([format])`**

Description Returns the text contained in the Clipboard. The format parameter, if specified, must be **`ebCFText`** (1). The **`format`** parameter must be either **`ebCFText`** or **`ebCFUnicodeText`**. If the **`format`** parameter is omitted, then the compiler first looks for text of the specified type depending on the platform:

Platform	Clipboard Format
Windows NT	UNICODE
Windows 98/Me	MBCS

Example

```
Option Compare Text
Sub Main
  If Clipboard.GetFormat(1) Then
    If Instr(Clipboard.GetText(1),"total",1) = 0 Then
      Session.Echo "The Clipboard doesn't contain the word ""total."""
    Else
      Session.Echo "The Clipboard contains the word ""total""."
    End If
  Else
    Session.Echo "The Clipboard does not contain text."
  End If
End Sub
```

See Also Clipboard$ (function); Operating System Control on page 15

Clipboard.SetText

Syntax **`Clipboard.SetText data$ [,format]`**

Description Copies the specified text string to the Clipboard. The **`data$`** parameter specifies the text to be copied to the Clipboard. The **`format`** parameter, if specified, must be **`ebCFText (1)`**. The **`format`** parameter must be either **`ebCFText`** or **`ebCFUnicodeText`**. If the **`format`** parameter is omitted, then the compiler places the text into the clipboard in the following format depending on the platform:

Platform	Clipboard Format
Windows NT	UNICODE
Windows 98/Me	MBCS

Example

```
Sub Main
  If Not Clipboard.GetFormat(1) Then Exit Sub
  Clipboard.SetText UCase$(Clipboard.GetText(1)),1
End Sub
```

See Also Clipboard$ (function); Operating System Control on page 15

CLng

Syntax `CLng(expression)`

Description Converts **expression** to a **Long**. This function accepts any expression convertible to a **Long**, including strings. A runtime error is generated if **expression** is **Null**. **Empty** is treated as 0. The passed expression must be within the following range:

`-2147483648 <= expression <= 2147483647`

A runtime error results if the passed expression is not within the above range.

When passed a numeric expression, this function has the same effect as assigning the numeric expression to a **Long**. Note that long variables are rounded before conversion.

When used with variants, this function guarantees that the expression is converted to a long variant (**VarType** 3).

Example This example displays the results for various conversions of **i** and **j** (note rounding).

```
Sub Main
    i% = 100
    j& = 123.666
    Session.Echo "The result is: " & CLng(i% * j&)  'Displays 12367.
    Session.Echo "The variant type is: " & Vartype(CLng(i%))
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Close

Syntax `Close [[#] filenumber [,[#] filenumber]...]`

Description Closes the specified files. If no arguments are specified, then all files are closed.

Example

```
Sub Main
  Open "test1" For Output As #1
  Open "test2" For Output As #2
  Open "test3" For Random As #3
  Open "test4" For Binary As #4
  Session.Echo "The next available file number is :" & FreeFile()
  Close #1        'Closes file 1 only.
  Close #2, #3      'Closes files 2 and 3.
```

```
  Close        'Closes all remaining files(4).
  Session.Echo "The next available file number is :" & FreeFile()
End Sub
```

See Also Drive, Folder, and File Access on page 4

ComboBox

Syntax **`ComboBox x,y,width,height,ArrayVariable,.Identifier`**

Description Defines a combo box within a dialog template. When the dialog is invoked, the combo box will be filled with the elements from the specified array variable. This statement can only appear within a dialog template (i.e., between the **`Begin Dialog`** and **`End Dialog`** statements). The **`ComboBox`** statement requires the following parameters:

Parameter	Description
`x`, **`y`**	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
`width`, **`height`**	Integer coordinates specifying the dimensions of the control in dialog units.
`ArrayVariable`	Single-dimensioned array used to initialize the elements of the combo box. If this array has no dimensions, then the combo box will be initialized with no elements. A runtime error results if the specified array contains more than one dimension. **`ArrayVariable`** can specify an array of any fundamental data type (structures are not allowed). Null and empty values are treated as zero-length strings.
`.Identifier`	Name by which this control can be referenced by statements in a dialog function (such as **`DlgFocus`** and **`DlgEnable`**). This parameter also creates a string variable whose value corresponds to the content of the edit field of the combo box. This variable can be accessed using the syntax: **`DialogVariable.Identifier.`**

When the dialog is invoked, the elements from **`ArrayVariable`** are placed into the combo box. The **`.Identifier`** variable defines the initial content of the edit field of the combo box. When the dialog is dismissed, the **`.Identifier`** variable is updated to contain the current value of the edit field.

Example

```
Sub Main
  Dim days$(6)
  days$(0) = "Monday"
  days$(1) = "Tuesday"
  days$(2) = "Wednesday"
  days$(3) = "Thursday"
  days$(4) = "Friday"
  days$(5) = "Saturday"
  days$(6) = "Sunday"
  Begin Dialog DaysDialogTemplate 16,32,124,96,"Days"
    OKButton 76,8,40,14,.OK
    Text 8,10,39,8,"&Weekdays:"
    ComboBox 8,20,60,72,days$,.Days
```

```
  End Dialog
  Dim DaysDialog As DaysDialogTemplate
  DaysDialog.Days = "Tuesday"
  r% = Dialog(DaysDialog)
  Session.Echo "You selected: " & DaysDialog.Days
End Sub
```

See Also User Interaction on page 16

Comments (topic)

Comments can be added to macro code in the following manner:

- All text between a single quotation mark and the end of the line is ignored:

```
Session.Echo "Hello"        'Displays a message box.
```

- The **REM** statement causes the compiler to ignore the entire line:

```
REM This is a comment.
```

- You can also use C-style multiline comment blocks /*...*/, as follows:

```
Session.Echo "Before comment"
/* This stuff is all commented out.
This line, too, will be ignored.
This is the last line of the comment. */
Session.Echo "After comment"
```

Note C-style comments can be nested.

See Also Keywords, Data Types, Operators, and Expressions on page 6; Macro Control and Compilation on page 10

Comparison Operators (topic)

Syntax **expression1 [< | > | <= | >= | <> | =] expression2**

Description Returns **True** or **False** depending on the operator. The comparison operators are listed in the following table:

Operator	Returns True If
>	**expression1** is greater than **expression2**
<	**expression1** is less than **expression2**
<=	**expression1** is less than or equal to **expression2**
>=	**expression1** is greater than or equal to **expression2**
<>	**expression1** is not equal to **expression2**
=	**expression1** is equal to **expression2**

This operator behaves differently depending on the types of the expressions, as shown in the following table:

Expression One	Expression Two	Result
Numeric	Numeric	Numeric comparison (see below).
String	String	String comparison (see below).
Numeric	String	Compile error.
Variant	String	String comparison (see below).
Variant	Numeric	Variant comparison (see below).
Null variant	Any data type	Null.
Variant	Variant	Variant comparison (see below).

String comparisons

If the two expressions are strings, then the operator performs a text comparison between the two string expressions, returning **`True`** if **`expression1`** is less than **`expression2`**. The text comparison is case-sensitive if **`Option Compare`** is **`Binary`**; otherwise, the comparison is case-insensitive.

When comparing letters with regard to case, lowercase characters in a string sort greater than uppercase characters, so a comparison of "a" and "A" would indicate that "a" is greater than "A".

Numeric comparisons

When comparing two numeric expressions, the less precise expression is converted to be the same type as the more precise expression.

Dates are compared as doubles. This may produce unexpected results as it is possible to have two dates that, when viewed as text, display as the same date when, in fact, they are different. This can be seen in the following example:

```
Sub Main
  Dim date1 As Date
  Dim date2 As Date
  date1 = Now
  date2 = date1 + 0.000001    'Adds a fraction of a second.
  Session.Echo date2 = date1  'Prints False (the dates are different).
  Session.Echo date1 & "," & date2  'Prints two dates that arethe same.
End Sub
```

Variant comparisons

When comparing variants, the actual operation performed is determined at execution time according to the following table:

Variant One	Variant Two	Result
Numeric	Numeric	Numeric comparison.
String	String	String comparison.
Numeric	String	Number less than string.
Null	Any other data type	Null.
Numeric	Empty	Compares number to 0.
String	Empty	Compares string to a zero-length string.

Examples

```
Sub Main
  'Tests two literals and displays the result.
  If 5 < 2 Then
    Session.Echo "5 is less than 2."
  Else
    Session.Echo "5 is not less than 2."
  End If
  'Tests two strings and displays the result.
  If "This" < "That" Then
    Session.Echo "'This' is less than 'That'."
  Else
    Session.Echo "'That' is less than 'This'."
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Const

Syntax **`Const name [As type] = expression [,name [As type] = expression]...`**

Description Declares a constant for use within the current macro. The **`name`** is only valid within the current macro. Constant names must follow these rules:

- Must begin with a letter.
- May contain only letters, digits, and the underscore character.
- Must not exceed 80 characters in length.
- Cannot be a reserved word.

Constant names are not case-sensitive. The **`expression`** must be assembled from literals or other constants. Calls to functions are not allowed except calls to the **`Chr$`** function, as shown below:

```
Const s$ = "Hello, there" + Chr(44)
```

Constants can be given an explicit type by declaring the **`name`** with a type-declaration character, as shown below:

```
Const a% = 5      'Constant Integer whose value is 5
Const b# = 5      'Constant Double whose value is 5.0
Const c$ = "5"    'Constant String whose value is "5"
Const d! = 5      'Constant Single whose value is 5.0
Const e& = 5      'Constant Long whose value is 5
```

The type can also be given by specifying the **`As`** type clause:

```
Const a As Integer = 5  'Constant Integer whose value is 5
Const b As Double = 5  'Constant Double whose value is 5.0
Const c As String = "5"  'Constant String whose value is "5"
Const d As Single = 5  'Constant Single whose value is 5.0
Const e As Long = 5  'Constant Long whose value is 5
```

You cannot specify both a type-declaration character and the **`type`**:

```
Const a% As Integer = 5  'THIS IS ILLEGAL.
```

If an explicit type is not given, then the compiler chooses the most imprecise type that completely represents the data, as shown below:

```
Const a = 5      'Integer constant
Const b = 5.5      'Single constant
Const c = 5.5E200    'Double constant
```

Constants defined within a **`Sub`** or **`Function`** are local to that subroutine or function. Constants defined outside of all subroutines and functions can be used anywhere within that macro. The following example demonstrates the scoping of constants:

```
Const DefFile = "default.txt"

Sub Test1
  Const DefFile = "foobar.txt"
  Session.Echo DefFile            'Displays "foobar.txt".
End Sub

Sub Test2
  Session.Echo DefFile            'Displays "default.txt".
End Sub
```

Example

```
Const crlf = Chr$(13) + Chr$(10)

Const s$ As String = "This is a constant."
Sub Main
  Session.Echo s$ & crlf & "The constants are shown above."
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Constants (topic)

Constants are variables that cannot change value during macro execution. You can define your own constants using the **Const** statement; preprocessor constants are defined using **#Const**. The following constants are predefined by the compiler.

Application State Constants

Constant	Value	Description
ebMinimized	1	The application is minimized.
ebMaximized	2	The application is maximized.
ebRestored	3	The application is restored.

Application.WindowState, Session.WindowState

Constant	Value	Description
smlMINIMIZE	0	The window is minimized.
smlRESTORE	1	The window is restored.
smlMAXIMIZE	2	The window is maximized.

Character Constants

Constant	Value	Description
ebBack	**Chr$(8)**	String containing a backspace.
ebCr	**Chr$(13)**	String containing a carriage return.
ebCrLf	**Chr$(13) & Chr$(10)**	String containing a carriage-return linefeed pair.
ebFormFeed	**Chr$(11)**	String containing a form feed.
ebLf	**Chr$(10)**	String containing a line feed.
ebNullChar	**Chr$(0)**	String containing a single null character.
ebNullString	0	Special string value used to pass null pointers to external routines.
ebTab	**Chr$(9)**	String containing a tab.
ebVerticalTab	**Chr$(12)**	String containing a vertical tab.

Circuit.SerialFlowControl

Constant	Value	Description
smlNOFLOWCONTROL	0	No flow control.

Constant	Value	Description
smlXONXOFF	1	XON/XOFF flow control.
smlRTSCTS	2	RTS/CTS flow control.
smlDTRDSR	3	DTR/DSR flow control.

Circuit.SerialParity

Constant	Value	Description
smlNOPARITY	0	No parity.
smlODDPARITY	1	Odd parity.
smlEVENPARITY	2	Even parity.
smlMARKPARITY	3	Mark parity.
smlSPACEPARITY	4	Space parity.

Circuit.TelnetBreakMode

Constant	Value	Description
smlBREAK	0	Set the breakmode to break.
SmlINTERRUPT	1	Set the breakmode to interrupt.

Circuit.TelnetCharacterMode

Constant	Value	Description
smlASCII	0	Set the character mode to ASCII.
smlBINARY	1	Set the character mode to binary.

Clipboard Constants

Constant	Value	Description
ebCFText	1	Text.
ebCFBitmap	2	Bitmap.
ebCFMetafile	3	Metafile.
ebCFDIB	8	Device-independent bitmap.
ebCFPalette	9	Palette.
ebCFUnicode	13	Unicode text.

Compiler Constants

Constant	Value
`Win32`	True
`Empty`	Empty
`False`	False
`Null`	Null
`True`	True

Date Constants

Constant	Value	Description
`ebUseSunday`	0	Use the date setting as specified by the current locale.
`ebSunday`	1	Sunday.
`ebMonday`	2	Monday.
`ebTuesday`	3	Tuesday.
`ebWednesday`	4	Wednesday.
`ebThursday`	5	Thursday.
`ebFriday`	6	Friday.
`ebSaturday`	7	Saturday.
`ebFirstJan1`	1	Start with week in which January 1 occurs.
`ebFirstFourDays`	2	Start with first week with at least four days in the new year.
`ebFirstFullWeek`	3	Start with first full week of the year.

File Constants

Constant	Value	Description
`ebNormal`	0	Read-only, archive, subdir, and none.
`ebReadOnly`	1	Read-only files.
`ebHidden`	2	Hidden files.
`ebSystem`	4	System files.
`ebVolume`	8	Volume labels.
`ebDirectory`	16	Subdirectory.
`ebArchive`	32	Files that have changed since the last backup.
`ebNone`	64	Files with no attributes.

File Type Constants

Constant	Value	Description
ebDOS	1	A DOS executable file.
ebWindows	2	A Windows executable file.

Font Constants

Constant	Value	Description
ebRegular	1	Normal font (i.e., neither bold nor italic).
ebItalic	2	Italic font.
ebBold	4	Bold font.
ebBoldItalic	6	Bold-italic font.

IMEStat Constants

Constant	Value	Description
ebIMENoOp	0	IME not installed.
ebIMEOn	1	IME on.
ebIMEOff	2	IME off.
ebIMEDisabled	3	IME disabled.
ebIMEHiragana	4	Hiragana double-byte character.
ebIMEKatakanaDbl	5	Katakana double-byte characters.
ebIMEKatakanaSng	6	Katakana single-byte characters.
ebIMEAlphaDbl	7	Alphanumeric double-byte characters.
ebIMEAlphaSng	8	Alphanumeric single-byte characters.

Math Constants

Constant	Value	Description
PI	3.1415...	Value of PI.

Session.EventWait

Constant	Value	Description
smlWAITSUCCESS	1	Successful match.
smlWAITTIMEOUT	-1	Timeout.
smlWAITMAXEVENTS	-2	Maximum events seen.
smlWAITERROR	-15	Miscellaneous error.

MsgBox Constants

Constant	Value	Description
`ebOKOnly`	0	Displays only the OK button.
`ebOKCancel`	1	Displays OK and Cancel buttons.
`ebAbortRetryIgnore`	2	Displays Abort, Retry, and Ignore buttons.
`ebYesNoCancel`	3	Displays Yes, No, and Cancel buttons.
`ebYesNo`	4	Displays Yes and No buttons.
`ebRetryCancel`	5	Displays Cancel and Retry buttons.
`ebCritical`	16	Displays the stop icon.
`EbQuestion`	32	Displays the question icon.
`EbExclamation`	48	Displays the exclamation icon.
`EbInformation`	64	Displays the information icon.
`EbApplicationModal`	0	The current application is suspended until the dialog is closed.
`EbDefaultButton1`	0	First button is the default button.
`EbDefaultButton2`	256	Second button is the default button.
`EbDefaultButton3`	512	Third button is the default button.
`EbSystemModal`	4096	All applications are suspended until the dialog is closed.
`EbOK`	1	Returned from MsgBox indicating that OK was pressed.
`EbCancel`	2	Returned from MsgBox indicating that Cancel was pressed.
`EbAbort`	3	Returned from MsgBox indicating that Abort was pressed.
`EbRetry`	4	Returned from MsgBox indicating that Retry was pressed.
`EbIgnore`	5	Returned from MsgBox indicating that Ignore was pressed.
`ebYes`	6	Returned from MsgBox indicating that Yes was pressed.
`ebNo`	7	Returned from MsgBox indicating that No was pressed.

Session.Capture File Handling

Constant	Value	Description
`smlOVERWRITE`	0	Overwrite an existing file.
`smlAPPEND`	1	Append to an existing file.
`smlPROMPTOVAPP`	2	Prompt whether to overwrite or append.

Session.KeyWait, Session.Collect

Constant	Value	Description
`smlWAITSUCCESS`	1	Successful match.
`smlWAITTIMEOUT`	-1	Timeout.
`smlWAITMAXCHARS`	-2	Maximum chars seen.
`smlWAITERROR`	-15	Miscellaneous error.

Session.StringWait

Constant	Value	Description
`smlWAITSUCCESS`	>=1	Successful match.
`smlWAITTIMEOUT`	-1	Timeout.
`smlWAITMAXCHARS`	-2	Maximum chars seen.
`smlWAITERROR`	-15	Miscellaneous error.

Session.ConfigInfo

Constant	Value	Description
`smlSESSIONPATH`	0	Full path of the SmarTerm session (STW) file.
`smlINSTALLPATH`	2	Full path to where SmarTerm is installed.

Session.EmulationInfo

Constant	Value	Description
`smlEMUFAMILY`	0	The emulation family.
`smlEMULEVEL`	1	The emulation level.

Session.KeyWait

Constant	Value	Description
`smlKEYWEXACT`	1	Non-case folded character/ASCII code
`smlKEYWNONEXACT`	2	Non-case folded character/ASCII code
`smlKEYWSCAN`	3	PC scan code
`smlKEYWVIRTUAL`	4	Virtual key code (Windows specific)
`smlKEYWDECKEY`	5	Emulation specific key code (DECKEY in PSL)
`smlKEYWBUTTON`	6	Locator button
`smlKEYWCOUNT`	7	Any key, (Use the count)

Session.Language, Application.InstalledLanguages, Application.StartupLanguage

Constant	Value	Description
smlGERMAN	1031	German.
smlENGLISH	1033	English.
smlFRENCH	1036	French.
smlSPANISH	1034	Spanish.

Shell Constants

Constant	Value	Description
ebHide	0	Application is initially hidden.
ebNormalFocus	1	Application is displayed at the default position and has the focus.
ebMinimizedFocus	2	Application is initially minimized and has the focus.
ebMaximizedFocus	3	Application is maximized and has the focus.
ebNormalNoFocus	4	Application is displayed at the default position and does not have the focus.
ebMinimizedNoFocus	6	Application is minimized and does not have the focus.

Macro Language Constants

Constant	Value	Description
True	-1	Boolean value True.
False	0	Boolean value False.
Empty	Empty	Variant of type 0, indicating that the variant is uninitialized.
Nothing	0	Value indicating that an object variable no longer references a valid object.
Null	Null	Variant of type 1, indicating that the variant contains no data.

String Conversion Constants

Constant	Value	Description
ebUpperCase	1	Converts string to uppercase.
ebLowerCase	2	Converts string to lowercase.
ebProperCase	3	Capitalizes the first letter of each word.
ebWide	4	Converts narrow characters to wide characters.
ebNarrow	8	Converts wide characters to narrow characters.

Constant	Value	Description
ebKatakana	16	Converts Hiragana characters to Katakana characters.
ebHiragana	32	Converts Katakana characters to Hiragana characters.
ebUnicode	64	Converts string from MBCS to UNICODE.
ebFromUnicode	128	Converts string from UNICODE to MBCS.

Variant Constants

Description	Constant	Value
ebEmpty	0	Variant has not been initialized.
ebNull	1	Variant contains no valid data.
ebInteger	2	Variant contains an integer.
ebLong	3	Variant contains a long.
ebSingle	4	Variant contains a single.
ebDouble	5	Variant contains a double.
ebCurrency	6	Variant contains a currency.
ebDate	7	Variant contains a date.
ebString	8	Variant contains a string.
ebObject	9	Variant contains an Object.
ebError	10	Variant contains an Error.
ebBoolean	11	Variant contains a boolean.
ebVariant	12	Variant contains an array of variants.
ebDataObject	13	Variant contains a data object.
ebArray	8192	Added to any of the other types to indicate an array of that type.

Cos

Syntax **Cos(number)**

Description Returns a **Double** representing the cosine of **number**. The **number** parameter is a **Double** specifying an angle in radians.

Example

```
Sub Main
    c# = Cos(3.14159 / 4)
    Session.Echo "The cosine of 45 degrees is: " & c#
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

CreateObject

Syntax **`CreateObject(class)`**

Description Creates an OLE Automation object and returns a reference to that object. The **`class`** parameter specifies the application used to create the object and the type of object being created. It uses the following syntax:

`"application.class",`

where **`application`** is the application used to create the object and **`class`** is the type of the object to create.

At runtime, **`CreateObject`** looks for the given application and runs that application if found. Once the object is created, its properties and methods can be accessed using the dot syntax (e.g., **`object.property`** = **`value`**).

There may be a slight delay when an automation server is loaded (this depends on the speed with which a server can be loaded from disk). This delay is reduced if an instance of the automation server is already loaded.

Examples This example uses CreateObject to instantiate a Visio object. It then uses the resulting object to create a new document.

```
Sub Main
    Dim Visio As Object
    Dim doc As Object
    Dim page As Object
    Dim shape As Object
    Set Visio = CreateObject("visio.application")
    'Create Visio object.
    Set doc = Visio.Documents.Add("")    'Create a new doc.
    Set page = doc.Pages(1)        'Get first page.
    Set shape = page.DrawRectangle(1,1,4,4)
    shape.text = "Hello, world."      'Set text within shape.
End Sub
```

See Also Objects on page 18; DDE Access on page 19

CSng

Syntax **`CSng(expression)`**

Description Converts **`expression`** to a **`Single`**. This function accepts any expression convertible to a **`Single`**, including strings. A runtime error is generated if **`expression`** is **`Null`**. **`Empty`** is treated as 0.0. A runtime error results if the passed expression is not within the valid range for **`Single`**.

When passed a numeric expression, this function has the same effect as assigning the numeric expression to a **`Single`**.

When used with variants, this function guarantees that the expression is converted to a **`Single`** variant (**`VarType`** 4).

Example

```
Sub Main
  s$ = "100"
  Session.Echo "The single value is: " & CSng(s$)
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

CStr

Syntax **`CStr(expression)`**

Description Converts **`expression`** to a **`String`**. Unlike **`Str$`** or **`Str`**, the string returned by **`CStr`** will not contain a leading space if the expression is positive. Further, the **`CStr`** function correctly recognizes thousands and decimal separators for your locale. Different data types are converted to **`String`** in accordance with the following rules:

Data Type	CStr Returns
Any numeric type	A string containing the number without the leading space for positive values
Date	A string converted to a date using the short date format
Boolean	A string containing either "True" or "False"
Null variant	A runtime error
Empty variant	A zero-length string

Example

```
Sub Main
  s# = 123.456
  Session.Echo "The string value is: " & CStr(s#)
End Sub
```

See Also Character and String Manipulation on page 3; Keywords, Data Types, Operators, and Expressions on page 6

CurDir, CurDir$

Syntax **`CurDir[$][(drive)]`**

Description Returns the current directory on the specified drive. If no **`drive`** is specified or **`drive`** is zero-length, then the current directory on the current drive is returned. **`CurDir$`** returns a **`String`**, whereas **`CurDir`** returns a **`String`** variant. There is a runtime error if **`drive`** is invalid.

Example

```
Const crlf = Chr$(13) + Chr$(10)
Sub Main
  save$ = CurDir$
  ChDir ("..")
```

```
  Session.Echo "Old directory: " & save$ & crlf & "New directory: " & CurDir$
  ChDir (save$)
  Session.Echo "Directory restored to: " & CurDir$
End Sub
```

See Also Drive, Folder, and File Access on page 4

Currency (data type)

Syntax **`Currency`**

Description Use to declare variables capable of holding fixed-point numbers with 15 digits to the left of the decimal point and 4 digits to the right. **`Currency`** variables are used to hold numbers within the following range:

```
-922,337,203,685,477.5808 <= currency <= 922,337,203,685,477.5807
```

Due to their accuracy, **`Currency`** variables are useful within calculations involving money.

The type-declaration character for **`Currency`** is @.

Internally, currency values are 8-byte integers scaled by 10000. Thus, when appearing within a structure, currency values require 8 bytes of storage. When used with binary or random files, 8 bytes of storage are required.

See Also Keywords, Data Types, Operators, and Expressions on page 6

CVar

Syntax **`CVar(expression)`**

Description Converts **`expression`** to a **`Variant`**.

Note Use of this function is not required because assignment to variant variables automatically performs the necessary conversion:

```
Sub Main()
  Dim v As Variant
  v = 4 & "th"          'Assigns "4th" to v.
  Session.Echo "You came in: " & v
  v = CVar(4 & "th")        'Assigns "4th" to v.
  Session.Echo "You came in: " & v
End Sub
```

Example

```
Sub Main
  Dim s As String
  Dim a As Variant
  s = CStr("The quick brown fox ")
```

```
  mesg = CVar(s & "jumped over the lazy dog.")
  Session.Echo mesg
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

CVErr

Syntax `CVErr(expression)`

Description This function converts an expression into a user-defined error number. A runtime error is generated under the following conditions:

- If **`expression`** is **`Null`**.
- If **`expression`** is a number outside the legal range for errors, which is as follows:

  `0 <= expression <= 65535`
- If **`expression`** is boolean.
- If **`expression`** is a **`String`** that can't be converted to a number within the legal range.

`Empty` is treated as 0.

Example

```
Sub Main
  Session.Echo "The error is: " & CStr(CVErr(2046))
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

D

Date (data type)

Syntax `Date`

Description Is capable of holding date and time values. **`Date`** variables are used to hold dates within the following range:

January 1, 100 00:00:00 <= **`date`** <= December 31, 9999 23:59:59

-6574340 <= **`date`** <= 2958465.99998843

Internally, dates are stored as 8-byte IEEE double values. The integer part holds the number of days since midnight, December 30, 1899, and the fractional part holds the number of seconds as a fraction of the day. For example, the number 32874.5 represents January 1, 1990 at 12:00:00.

When appearing within a structure, dates require 8 bytes of storage. Similarly, when used with binary or random files, 8 bytes of storage are required.

There is no type-declaration character for **`Date`**.

Date variables that haven't been assigned are given an initial value of 0 (i.e., December 30, 1899).

Date literals

Literal dates are specified using pound signs:

```
Dim d As Date
d = #January 1, 1990#
```

The interpretation of the date string (i.e., January 1, 1990 in the above example) occurs at runtime, using the current country settings. This is a problem when interpreting dates such as 1/2/1990. If the date format is M/D/Y, then this date is January 2, 1990. If the date format is D/M/Y, then this date is February 1, 1990. To remove any ambiguity when interpreting dates, use the universal date format:

```
date_variable = #YY/MM/DD HH:MM:SS#
```

The following example specifies the date June 3, 1965, using the universal date format:

```
Dim d As Date
d = #1965/6/3 10:23:45#
```

Dates and Year 2000 Calculations

The Date object in Persoft's macro language always stores the year with 4 digits, regardless of how the date was entered. However, if a year is specified with only two digits, and that year is less than 30, then the macro language assumes a twenty-first century date. Otherwise, it assumes a twentieth-century date. In pseudocode, the decision looks like this:

```
If 0 < two-digit year < 30 Then
   year = 2000 + two-digit year
Else
   year = 1900 + two-digit year
End If
```

For example, if you specify the date 1/1/29, the macro language stores it as 1/1/2029 and all calculations will assume the year to be 2029: However, if you specify the date 1/1/30, then the macro language stores it as 1/1/1930.

Compensating for dates specifying two-digit years

Because the macro language calculates years correctly given four-digit dates, our recommendation is that at all times dates in your macros specify the year with four digits. Ensuring that this is the case may require you to revise your macros if one or more date sources specify two-digit years. There are three possible sources for dates specifying two-digit years:

- Date literals (such as `#1/1/24#`)
- Macro input routines that allow users to specify two-digit years
- Legacy data in a source that contains dates specifying two-digit years

Date literals

If you have date literals specifying two-digit years, the solution is simple: revise the macros to specify all four digits of years in the date literals. Since date literals are marked off on either end with the pound (`#`) character, it's easy to use the Macro Editor or any ASCII text editor to search macros for date literals.

For example, the following macro incorrectly sets the default startup date to 2029 by specifying the date literal with a two-digit year:

```
Sub testdate1
'!Example of the incorrect definition of a date literal
  Dim StartupDate#, DefaultStartupDate#
  DefaultStartupDate= #7/12/29#   'This is the problem definition
```

```
  ' Make sure that StartupDate is defined:
  ' Note that 12/30/1899 is the zero-point for dates.
  If StartupDate# = 0 Then
     MsgBox "StartupDate= " & Format(StartupDate#, "long date")
     StartupDate#= DefaultStartupDate#
  End If

  MsgBox "StartupDate= " & Format(StartupDate#, "long date")
End Sub
```

This macro has a routine that makes sure that `StartupDate#` is at least set to a default value before later performing operations on it. Unfortunately, the default value (`DefaultStartupDate#`) is not clearly specified with a four-digit year. You might not catch this error unless the StartupDate# variable was undefined for some reason, and so became set to 7/12/2029. To correct this error, search through your macros and make sure that date literals specify all four digits for the year:

```
Sub testdate2
'!Example of the correct definition of a date literal
  Dim StartupDate#, DefaultStartupDate#
  DefaultStartupDate= #7/12/1929#   'This is the corrected definition

  ' Make sure that StartupDate is defined:
  ' Note that 12/30/1899 is the zero-point for dates.
  If StartupDate# = 0 Then
     MsgBox "StartupDate= " & Format(StartupDate#, "long date")
     StartupDate#= DefaultStartupDate#
  End If

  MsgBox "StartupDate= " & Format(StartupDate#, "long date")
End Sub
```

Date input

If you have macro input routines that allow users to specify two-digit years, the solution is to revise the macros to check for four-digit years, forcing the user to re-specify the date if they fail to comply. The following code fragment provides a simple check (although it does not check for other input errors).

```
Sub testdate3
'! Example showing how to check for a 4-digit year in user input.
   Dim strDate$, strMonth$, strDay$, strYear$, EnteredDate#

   Do While len(strYear$) < 4  'Loop until the year has 4 digits:
      StrDate$= InputBox("Enter date (MM/DD/YYYY): ", "Date Converted")

      If StrDate$ = "" Then  'Clicked OK without entering a date,
         Exit Sub            'so we quit the macro
      End If

      'Parse each item in the date
      strMonth$ = Item$(strDate$, 1, 1, "/")
      strDay$ = Item$(strDate$, 2, 2, "/")
      strYear$ = Item$(strDate$, 3, 3, "/")
   Loop

   'OK, the year finally has 4 digits. Confirm the date:
```

```
   EnteredDate# = CDate(strDate$)
   MsgBox "Date entered: " & strDate$

End Sub
```

When you run this macro, an input box appears asking for the date and indicating the correct format. If you click OK without entering anything, the macro ends. Otherwise, it loops as long as the year has fewer than four digits, redisplaying the input box for a correct date. When the macro detects that the year has been correctly entered, then it displays a message box confirming the date.

Legacy data

If you have legacy data in a source that specifies dates using only two digits for the year, which cannot be changed to specify four digits for the year, and you anticipate adding new data to that source, your macros will have to compensate. How you compensate will depend upon what kind of date information is being stored, and what operations you need to perform on the dates.

For example, if you need to calculate the span of years between a date stored in the database and today, and you know that a negative timespan would be an error, you can test for a negative timespan and then correct it if it occurs. The following code fragment provides a simple example.

```
Sub testdate4
'!Example showing how to correct for 2-digit dates in legacy data

   Dim date1 As Date
   Dim date2 As Date
   Dim diff As Date
   date1 = #1/1/24#  'This date would come from the database
   date2 = Date      'This is the current date

'Now calculate the elapsed years: date2 - date1
   diff = DateDiff("yyyy",date1,date2)
   MsgBox "The raw date difference is: " & CDbl(diff) & " years."

'Now run the correction routine. If the elapsed timeperiod is negative, then
'subtract a century from date1 and recalculate. Otherwise, everything is fine.
   If CInt(diff)<0 Then
      date1= DateAdd("yyyy", -100, date1)
      MsgBox "The corrected date1 year is: " & DatePart("yyyy", date1)
      diff = DateDiff("yyyy",date1, date2)
      MsgBox "The corrected date difference is " & CDbl(diff) & " years."
   Else
      MsgBox "The date difference, " & CDbl(diff) & " years, was correct."
   End if

End Sub
```

This macro first calcuates the number of years between **`date1#`** and **`date2#`**. If the result is negative, then the macro subtracts a century from **`date1#`** and recalculates the difference. To verify that the macro does not subtract a century from valid dates, replace the line defining **`date1#`** as **`#1/1/24#`** to define the year with four digits: **`#1/1/1924#`**.

See Also Keywords, Data Types, Operators, and Expressions on page 6; Time and Date Access on page 17

Date, Date$ (functions)

Syntax `Date[$][()]`

Description Returns the current system date. The **`Date$`** function returns the date using the short date format. The **`Date`** function returns the date as a **`Date`** variant.

Use the **`Date/Date$`** statements to set the system date.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  TheDate$ = Date$()
  Date$ = "01/01/95"
  Session.Echo "Saved date is: " & TheDate$ & crlf & "Changed date is: " & Date$()
  Date$ = TheDate$
  Session.Echo "Restored date to: " & TheDate$
End Sub
```

See Also Time and Date Access on page 17

Date, Date$ (statements)

Syntax `Date[$] = newdate`

Description Sets the system date to the specified date. The **`Date$`** statement requires a string variable using one of the following formats:

```
MM-DD-YYYY
MM-DD-YY
MM/DD/YYYY
MM/DD/YY,
```

where **`MM`** is a two-digit month between 1 and 31, **`DD`** is a two-digit day between 1 and 31, and **`YYYY`** is a four-digit year between 1/1/100 and 12/31/9999.

The **`Date`** statement converts any expression to a date, including string and numeric values. Unlike the **`Date$`** statement, **`Date`** recognizes many different date formats, including abbreviated and full month names and a variety of ordering options. If **`newdate`** contains a time component, it is accepted, but the time is not changed. An error occurs if **`newdate`** cannot be interpreted as a valid date.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  TheDate$ = Date$()
  Date$ = "01/01/95"
  Session.Echo "Saved date: " & TheDate$ & crlf & "Changed date: " & _
Date$()
  Date$ = TheDate$
  Session.Echo "Restored date to: " & TheDate$
End Sub
```

See Also Time and Date Access on page 17

DateAdd

Syntax **`DateAdd(interval, number, date)`**

Description Returns a **`Date`** variant representing the sum of date and a specified number (**`number`**) of time intervals (**`interval`**). This function adds a specified number (**`number`**) of time intervals (**`interval`**) to the specified date (**`date`**). The following table describes the named parameters to the **`DateAdd`** function:

Parameter	Description
`interval`	String expression indicating the time interval used in the addition.
`number`	Integer indicating the number of time intervals you wish to add. Positive values result in dates in the future; negative values result in dates in the past.
`date`	Any expression convertible to a date string expression. An example of a valid date/time string would be "January 1, 1993".

The **`interval`** parameter specifies what unit of time is to be added to the given date. It can be any of the following:

Time	Interval
`"y"`	Day of the year
`"yyyy"`	Year
`"d"`	Day
`"m"`	Month
`"q"`	Quarter
`"ww"`	Week
`"h"`	Hour
`"n"`	Minute
`"s"`	Second
`"w"`	Weekday

To add days to a date, you may use either day, day of the year, or weekday, as they are all equivalent (**`"d"`**, **`"y"`**, **`"w"`**).

The **`DateAdd`** function will never return an invalid date/time expression. The following example adds two months to December 31, 1992:

```
s# = DateAdd("m", 2, "December 31, 1992")
```

In this example, **s$** is returned as the double-precision number equal to "February 28, 1993", not "February 31, 1993".

There is a runtime error if you try subtracting a time interval that is larger than the time value of the date.

Example

```
Sub Main
  Dim sdate$
  sdate$ = Date$
  NewDate# = DateAdd("yyyy", 4, sdate$)
  NewDate# = DateAdd("m", 3, NewDate#)
  NewDate# = DateAdd("ww", 2, NewDate#)
  NewDate# = DateAdd("d", 1, NewDate#)
  s$ = "Four years, three months, two weeks, and one day from now: "
  s$ = s$ & Format(NewDate#, "long date")
  Session.Echo s$
End Sub
```

See Also Time and Date Access on page 17

DateDiff

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

Description Returns a **Date** variant representing the number of given time intervals between **date1** and **date2**. The following describes the named parameters:

Parameter	Description
interval	String expression indicating the specific time interval you wish to find the difference between. An error is generated if **interval** is null.
date1	Any expression convertible to a date. An example of a valid date/time string would be "January 1, 1994".
date2	Any expression convertible to a date. An example of a valid date/time string would be "January 1, 1994".
firstdayofweek	Indicates the first day of the week. If omitted, then Sunday is assumed (i.e., the constant ebSunday described below).
firstweekofyear	Indicates the first week of the year. If omitted, then the first week of the year is considered to be that containing January 1 (i.e., the constant ebFirstJan1 as described below).

The following lists the valid time interval strings and the meanings of each. The **Format$** function uses the same expressions

Time	Interval
`"y"`	Day of the year
`"yyyy"`	Year
`"d"`	Day
`"m"`	Month
`"q"`	Quarter
`"ww"`	Week
`"h"`	Hour
`"n"`	Minute
`"s"`	Second
`"w"`	Weekday

To find the number of days between two dates, you may use either day or day of the year, as they are both equivalent (**`"d"`**, **`"y"`**).

The time interval weekday (**`"w"`**) will return the number of weekdays occurring between **`date1`** and **`date2`**, counting the first occurrence but not the last. However, if the time interval is week (**`"ww"`**), the function will return the number of calendar weeks between **`date1`** and **`date2`**, counting the number of Sundays. If **`date1`** falls on a Sunday, then that day is counted, but if **`date2`** falls on a Sunday, it is not counted.

The **`firstdayofweek`** parameter, if specified, can be any of the following constants:

Constant	Value	Description
`ebUseSystem`	0	Use the system setting for **`firstdayofweek`**.
`ebSunday`	1	Sunday (the default)
`ebMonday`	2	Monday
`ebTuesday`	3	Tuesday
`ebWednesday`	4	Wednesday
`ebThursday`	5	Thursday
`ebFriday`	6	Friday
`ebSaturday`	7	Saturday

The **`firstdayofyear`** parameter, if specified, can be any of the following constants:

Constant	Value	Description
ebUseSystem	0	Use the system setting for **firstdayofyear**.
ebfirstjan1	1	The first week of the year is that in which January 1 occurs (the default).
ebfirstfourdays	2	The first week of the year is that containing at least four days in the year.
ebfirstfullweek	3	The first week of the year is the first full week of the year.

The **DateDiff** function will return a negative date/time value if **date1** is a date later in time than **date2**. If **date1** or **date2** are **Null**, then **Null** is returned.

Example

```
Sub Main
  today$ = Format(Date$,"Short Date")
  NextWeek = Format(DateAdd("d", 14, today$),"Short Date")
  DifDays# = DateDiff("d", today$, NextWeek)
  DifWeek# = DateDiff("w", today$, NextWeek)
  s$ = "The difference between " & today$ & " and " & NextWeek
  s$ = s$ & " is: " & DifDays# & " days or " & DifWeek# & " weeks"
  Session.Echo s$
End Sub
```

See Also Time and Date Access on page 17

DatePart

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

Description Returns an **Integer** representing a specific part of a date/time expression. The **DatePart** function decomposes the specified date and returns a given date/time element. The following table describes the named parameters:

Parameter	Description
interval	String expression that indicates the specific time interval you wish to identify within the given date.
date	Any expression convertible to a date. An example of a valid date/time string would be "January 1, 1995".
firstdayofweek	Indicates the first day of the week. If omitted, then Sunday is assumed (i.e., the constant **ebSunday** described below).
firstweekofyear	Indicates the first week of the year. If omitted, then the first week of the year is considered to be that containing January 1 (i.e., the constant **ebFirstJan1** as described bellow).

The following table lists the valid time interval strings and the meanings of each. The **Format$** function uses the same expressions.

Time	Interval
`"y"`	Day of the year
`"yyyy"`	Year
`"d"`	Day
`"m"`	Month
`"q"`	Quarter
`"ww"`	Week
`"h"`	Hour
`"n"`	Minute
`"s"`	Second
`"w"`	Weekday

The **`firstdayofweek`** parameter, if specified, can be any of the following constants:

Constant	Value	Description
`ebUseSystem`	0	Use the system setting for **`firstdayofweek`**.
`ebsunday`	1	Sunday (the default)
`ebMonday`	2	Monday
`ebTuesday`	3	Tuesday
`ebWednesday`	4	Wednesday
`ebThursday`	5	Thursday
`ebFriday`	6	Friday
`ebSaturday`	7	Saturday

The **`firstdayofyear`** parameter, if specified, can be any of the following constants:

Constant	Value	Description
`ebUseSystem`	0	Use the system setting for **`firstdayofyear`**.
`ebfirstjan1`	1	The first week of the year is that in which January 1 occurs (the default).
`ebfirstfourdays`	2	The first week of the year is that containing at least four days in the year.
`ebfirstfullweek`	3	The first week of the year is the first full week of the year.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  today$ = Date$
  qtr = DatePart("q",today$)
```

```
  yr = DatePart("yyyy",today$)
  mo = DatePart("m",today$)
  wk = DatePart("ww",today$)
  da = DatePart("d",today$)
  s$ = "Quarter: " & qtr & crlf
  s$ = s$ & "Year    : " & yr & crlf
  s$ = s$ & "Month    : " & mo & crlf
  s$ = s$ & "Week    : " & wk & crlf
  s$ = s$ & "Day    : " & da & crlf
  Session.Echo s$
End Sub
```

See Also Time and Date Access on page 17

DateSerial

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

Description Returns a **`Date`** variant representing the specified date. The **`DateSerial`** function takes the following named parameters:

Named Parameter	Description
`year`	Integer between 100 and 9999
`month`	Integer between 1 and 12
`day`	Integer between 1 and 31

Example

```
Sub Main
  tdate# = DateSerial(1993,08,22)
  Session.Echo "The DateSerial value for August 22, 1993, is: " & tdate#
End Sub
```

See Also Time and Date Access on page 17

DateValue

Syntax **`DateValue(date)`**

Description Returns a **`Date`** variant representing the date contained in the specified string argument.

Example

```
Sub Main
  tdate$ = Date$
  tday = DateValue(tdate$)
  Session.Echo tdate & " date value is: " & tday$
End Sub
```

See Also Time and Date Access on page 17

Day

Syntax **`Day(date)`**

Description Returns the day of the month specified by **`date`**. The value returned is an **`Integer`** between 0 and 31 inclusive. The **`date`** parameter is any expression that converts to a **`Date`**.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  CurDate = Now()
  Session.Echo "Today is day " & Day(CurDate) & " of the month." & _
crlf & _ "Tomorrow is day " & Day(CurDate + 1)
End Sub
```

See Also Time and Date Access on page 17

DDB

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

Description Calculates the depreciation of an asset for a specified **`period`** of time using the double-declining balance method. The double-declining balance method calculates the depreciation of an asset at an accelerated rate. The depreciation is at its highest in the first period and becomes progressively lower in each additional period. **`DDB`** uses the following formula to calculate the depreciation:

```
DDB =((Cost-Total_depreciation_from_all_other_periods) * 2)/Life
```

The **`DDB`** function uses the following named parameters:

Parameter	Description
`cost`	Double representing the initial cost of the asset
`salvage`	Double representing the estimated value of the asset at the end of its predicted useful life
`life`	Double representing the predicted length of the asset's useful life
`period`	Double representing the period for which you wish to calculate the depreciation
`factor`	Depreciation factor determining the rate the balance declines. If this parameter is missing, then 2 is assumed (double-declining method).

The **`life`** and **`period`** parameters must be expressed using the same units. For example, if **`life`** is expressed in months, then **`period`** must also be expressed in months.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  s$ = "Depreciation Table" & crlf & crlf
  For yy = 1 To 4
    CurDep# = DDB(10000.0,2000.0,10,yy)
    s$ = s$ & "Year " & yy & " : " & CurDep# & crlf
  Next yy
  Session.Echo s$
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

DDEExecute

Syntax **`DDEExecute channel, command$`**

Description Executes a command in another application. The **`DDEExecute`** statement takes the following parameters:

Parameter	Description
`channel`	Integer containing the DDE channel number returned from **`DDEInitiate`**. An error will result if **`channel`** is invalid.
`command$`	String containing the command to be executed. The format of **`command$`** depends on the receiving application.

If the receiving application does not execute the instructions, there is a runtime error.

Example This example selects a cell in an Excel spreadsheet.

```
Sub Main
  q$ = Chr(34)
  ch% = DDEInitiate("Excel","c:\sheets\test.xls")
  cmd$ = "[Select(" & q$ & "R1C1:R8C1" & q$ & ")]"
  DDEExecute ch%,cmd$
  DDETerminate ch%
End Sub
```

See Also DDE Access on page 19

DDEInitiate

Syntax **`DDEInitiate(application$, topic$)`**

Description Initializes a DDE link to another application and returns a unique number subsequently used to refer to the open DDE channel. The **`DDEInitiate`** statement takes the following parameters:

Parameter	Description
`application$`	String containing the name of the application (the server) with which a DDE conversation will be established.
`topic$`	String containing the name of the topic for the conversation. The possible values for this parameter are described in the documentation for the server application.

This function returns 0 if the compiler cannot establish the link. This will occur under any of the following circumstances:

- The specified application is not running.

- The topic was invalid for that application.
- Memory or system resources are insufficient to establish the DDE link.

Example This example selects a range of cells in an Excel spreadsheet.

```
Sub Main
  q$ = Chr(34)
  ch% = DDEInitiate("Excel","c:\sheets\test.xls")
  cmd$ = "[Select(" & q$ & "R1C1:R8C1" & q$ & ")]"
  DDEExecute ch%,cmd$
  DDETerminate ch%
End Sub
```

See Also DDE Access on page 19

DDEPoke

Syntax **`DDEPoke channel, DataItem, value`**

Description Sets the value of a data item in the receiving application associated with an open DDE link. The **`DDEPoke`** statement takes the following parameters:

Parameter	Description
`channel`	Integer containing the DDE channel number returned from DDEInitiate. An error will result if **`channel`** is invalid.
`DataItem`	Data item to be set. This parameter can be any expression convertible to a string. The format depends on the server.
`Value`	The new value for the data item. This parameter can be any expression convertible to a string. The format depends on the server. A runtime error is generated if **`value`** is null.

Example This example pokes a value into an Excel spreadsheet.

```
Sub Main
  ch% = DDEInitiate("Excel","c:\sheets\test.xls")
  DDEPoke ch%,"R1C1","980"
  DDETerminate ch%
End Sub
```

See Also DDE Access on page 19

DDERequest, DDERequest$

Syntax **`DDERequest[$](channel,DataItem$)`**

Description Returns the value of the given data item in the receiving application associated with the open DDE channel. **`DDERequest$`** returns a **`String`**, whereas **`DDERequest`** returns a **`String`** variant. The **`DDERequest/DDERequest$`** functions take the following parameters:

Parameter	Description
channel	Integer containing the DDE channel number returned from DDEInitiate. An error results if **channel** is invalid.
DataItem$	String containing the name of the data item to request. The format for this parameter depends on the server.

The format for the returned value depends on the server.

Example This example gets a value from an Excel spreadsheet.

```
Sub Main
  ch% = DDEInitiate("Excel","c:\excel\test.xls")
  s$ = DDERequest$(ch%,"R1C1")
  DDETerminate ch%
  Session.Echo s$
End Sub
```

See Also DDE Access on page 19

DDESend

Syntax **DDESend application$, topic$, DataItem, value**

Description Initiates a DDE conversation with the server as specified by **application$** and **topic$** and sends that server a new value for the specified item. The **DDESend** statement takes the following parameters:

Parameter	Description
application$	String containing the name of the application (the server) with which a DDE conversation will be established.
topic$	String containing the name of the topic for the conversation. The possible values for this parameter are described in the documentation for the server application.
DataItem	Data item to be set. This parameter can be any expression convertible to a string. The format depends on the server.
Value	New value for the data item. This parameter can be any expression convertible to a string. The format depends on the server. A runtime error is generated if **value** is null.

The **DDESend** statement performs the equivalent of the following statements:

```
ch% = DDEInitiate(application$, topic$)
DDEPoke ch%, item, data
DDETerminate ch%
```

Example This code sets the content of the first cell in an Excel spreadsheet.

```
Sub Main
  On Error Goto Trap1
  DDESend "Excel","c:\excel\test.xls","R1C1","Hello, world."
  On Error Goto 0
  'Add more lines here.
Exit Sub
Trap1:
  MsgBox "Error sending data to Excel."
End Sub
```

See Also DDE Access on page 19

DDETerminate

Syntax **`DDETerminate channel`**

Description Closes the specified DDE channel. The **`channel`** parameter is an **`Integer`** containing the DDE channel number returned from **`DDEInitiate`**. An error will result if **`channel`** is invalid. All open DDE channels are automatically terminated when the macro ends.

Example This code sets the content of the first cell in an Excel spreadsheet.

```
Sub Main
  q$ = Chr(34)
  ch% = DDEInitiate("Excel","c:\sheets\test.xls")
  cmd$ = "[Select(" & q$ & "R1C1:R8C1" & q$ & ")]"
  DDEExecute ch%,cmd$
  DDETerminate ch%
End Sub
```

See Also DDE Access on page 19

DDETerminateAll

Syntax **`DDETerminateAll`**

Description Closes all open DDE channels. All open DDE channels are automatically terminated when the macro ends.

Example This code selects the contents of the first cell in an Excel spreadsheet.

```
Sub Main
  q$ = Chr(34)
  ch% = DDEInitiate("Excel","c:\sheets\test.xls")
  cmd$ = "[Select(" & q$ & "R1C1:R8C1" & q$ & ")]"
  DDEExecute ch%,cmd$
  DDETerminateAll
End Sub
```

See Also DDE Access on page 19

DDETimeout

Syntax `DDETimeout milliseconds`

Description Sets the number of milliseconds that must elapse before any DDE command times out. The **`milliseconds`** parameter is a **`Long`** and must be within the following range:

```
0 <= milliseconds <= 2,147,483,647
```

The default is 10,000 (10 seconds).

Example

```
Sub Main
  q$ = Chr(34)
  ch% = DDEInitiate("Excel","c:\sheets\test.xls")
  DDETimeout(20000)
  cmd$ = "[Select(" & q$ & "R1C1:R8C1" & q$ & ")]"
  DDEExecute ch%,cmd$
  DDETerminate ch%
End Sub
```

See Also DDE Access on page 19

Declare

Syntax

```
Declare {Sub | Function} name[TypeChar] [{[ParameterList]}] [As type]
```

```
Declare {Sub | Function} name[TypeChar] [CDecl | Pascal | System |
StdCall] [Lib "LibName$" [Alias "AliasName$"]] [([ParameterList])] [As type]
```

The first syntax is for prototyping subroutines and functions for later portions of the macro or for other members of the macro collective, while the second syntax is for declaring compiled routines stored in external .DLL files. In both cases, **`ParameterList`** is a comma-separated list of the following (up to 30 parameters are allowed):

```
[Optional] [ByVal | ByRef] ParameterName[()] [As ParameterType]
```

Description **`Declare`** statements must appear outside of any **`Sub`** or **`Function`** declaration. **`Declare`** statements are only valid during the life of the macro in which they appear. The **`Declare`** statement uses the following parameters:

Parameter	Description
name	Any valid name. When you declare functions, you can include a type-declaration character to indicate the return type. This name is specified as a normal keyword— i.e., it does not appear within quotes.
TypeChar	An optional type-declaration character used when defining the type of data returned from functions. It can be any of the following characters: #, !, $, @, %, or &. For external functions, the @ character is not allowed. Type-declaration characters can only appear with function declarations, and take the place of the **As type** clause. Currency data cannot be returned from external functions. Therefore, the @ type-declaration character cannot be used when declaring external functions.
Decl	Optional keyword indicating that the external subroutine or function uses the C calling convention. With C routines, arguments are pushed right to left on the stack and the caller performs stack cleanup.
Pascal	Optional keyword indicating that this external subroutine or function uses the Pascal calling convention. With Pascal routines, arguments are pushed left to right on the stack and the called function performs stack cleanup.
System	Optional keyword indicating that the external subroutine or function uses the System calling convention. With System routines, arguments are pushed right to left on the stack, the caller performs stack cleanup, and the number of arguments is specified in the AL register.
StdCall	Optional keyword indicating that the external subroutine or function uses the StdCall calling convention. With StdCall routines, arguments are pushed right to left on the stack and the called function performs stack cleanup.

LibName$	Must be specified if the routine is stored in an external .DLL file. This parameter specifies the name of the library or code resource containing the external routine and must appear within quotes. The **LibName$** parameter can include an optional path specifying the exact location of the library or code resource. Alias name that must be given to provide the name of the routine if the **name** parameter is not the routine's real name. For example, the following two statements declare the same routine:
	
	`Declare Function GetCurrentTime Lib "user" () As Integer`
	
	`Declare Function GetTime Lib "user" Alias "GetCurrentTime" _As Integer`
	
	Use an alias when the name of an external routine conflicts with the name of an internal routine or when the external routine name contains invalid characters. The **AliasName$** parameter must appear within quotes.

type	Indicates the return type for functions. For external functions, the valid return types are: integer, long, string, single, double, date, boolean, and data objects. Currency, variant, fixed-length strings, arrays, OLE Automation objects, and user-defined types cannot be returned by external functions.

Parameter	Description
`Optional`	Keyword indicating that the parameter is optional. All optional parameters must be of type variant. Furthermore, all parameters that follow the first optional parameter must also be optional. If this keyword is omitted, then the parameter being defined is required when calling this subroutine or function.
`ByVal`	Optional keyword indicating that the caller will pass the parameter by value. Parameters passed by value cannot be changed by the called routine.
`ByRef`	Optional keyword indicating that the caller will pass the parameter by reference. Parameters passed by reference can be changed by the called routine. If neither ByVal or ByRef are specified, then ByRef is assumed.
`Parameter-Name`	Name of the parameter, which must follow naming conventions: Must start with a letter; may contain letters, digits, and the underscore character (_). Punctuation and type-declaration characters are not allowed. The exclamation point (!) can appear within the name as long as it is not the last character, in which case it is interpreted as a type-declaration character. Must not exceed 80 characters in length. Also, `ParameterName` can end with an optional type-declaration character specifying the type of that parameter (i.e., any of the following characters: %, &, !, #, @).
`()`	Indicates that the parameter is an array.
`Parameter-Type`	Specifies the type of the parameter (e.g., integer, string, variant, and so on). The As `ParameterType` clause should only be included if `ParameterName` does not contain a type-declaration character. In addition to the default data types, `ParameterType` can specify any user-defined structure, OLE Automation object, or data object . If the data type of the parameter is not known in advance, then the Any keyword can be used. This forces the compiler to relax type checking, allowing any data type to be passed in place of the given argument. For example: `Declare Sub Convert Lib "mylib" (a As Any)` The Any data type can only be used when passing parameters to external routines.

Prototyping macro subroutines and functions

Functions that need to be accessible to other members of the macro collective must be prototyped with the **`Declare`** statement. This prototyping is optional for subroutines unless you have also required explicit type-checking with the **`Option Explicit`** statement.

The following sample shows how to prototype subroutines and functions, and how to call those subroutines and functions from other macros in the collective. See “Modules and collectives” on page 32 for more information on which modules can provide subroutines and functions, and which modules can access them.

Adding and subtracting via prototypes

In this example, we create a small palette of SmarTerm Buttons that ask for two numbers and either add them or multiply them. Follow these steps:

1. Use the Tools>Macros command to add a subroutine called Add to the user macro file. The macro should look like this:

```
Sub Add(x As Double, y As Double)
   '! Add two numbers.
   Msgbox x & " plus " & y & " equals " & x + y
End Sub
```

2. While you have the user macro file open, add the following function after the Add subroutine.

```
Function Multiply(x As Double, y As Double) As Double
   'Multiply two numbers together.
   Multiply = x * y
End Function
```

   Then save and close the user macro file.

3. Now create a new palette of SmarTerm Buttons called Math. It should have two buttons, an Add button and a Multiply button.
4. Edit the Add button to attach an embedded macro called GetSum. GetSum should look like this:

```
Sub GetSum
   '! Add to numbers by calling Add( ) in the user macro file.
   Dim x As Double
   Dim y As Double
   x = InputBox("Enter the first number.", "Addition Example")
   y = InputBox("Enter the first number.", "Addition Example")

   Add x,y ’Using the Add subroutine in the user macro file

End Sub
```

   Save the macro and close the macro editor.

5. Now edit the Multiply button to attach an embedded macro called GetProduct. GetProduct should look like this:

```
Sub GetProduct
   'Multiply two numbers using the Multiply function in the user macro file
   Dim Product
   Dim x As Double
   Dim y As Double
   x = InputBox("Enter the first number.", "Multiplication Example")
   y = InputBox("Enter the first number.", "Multiplication Example")

   Product = Multiply(x,y) ’Using the Multiply function in the user macro file

   Msgbox x & " times " &  CStr(y) & " equals " & Product, ebOKOnly, "Muliplication"
End Sub
```

6. Don't save and close the macro file just yet. While you have this macro open, scroll to the top of the editor and insert the following lines to the very beginning of the file:

```
Option Explicit
Declare Sub Add(x As Double, y As Double)
Declare Function Multiply(x As Double, y as Double) As Double
```

   The first line sets the compiler to require type-checking. You must add this line to be able to access external functions. The next line prototypes the Add subroutine, and the third line prototypes the Multiply function.

7. Now save and close the macro file, save the palette and close the palette editor, and try out your new Buttons. You can confirm that subroutines are available without Option Explicit by commenting out the Option Explicit statement in the Buttons macro and then trying out the Buttons again. The Add Button will work, while the Multiply Button will halt with an error message.

Declaring routines in external .DLL files

The following sections describe some of the issues involved in calling routines stored in external .DLL files. This is a very powerful feature of the macro language, as it gives you access to any routine in any accessible .DLL file on the computer. However, because of differences in calling conventions and data representation, it can be tricky to implement.

Passing parameters

By default, the compiler passes arguments by reference. Many external routines require a value rather than a reference to a value. The **`ByVal`** keyword does this. For example, this C routine:

```
void MessageBeep(int);
```

would be declared as follows:

```
Declare Sub MessageBeep Lib "user" (ByVal n As Integer)
```

As an example of passing parameters by reference, consider the following C routine which requires a pointer to an integer as the third parameter:

```
int SystemParametersInfo(int,int,int *,int);
```

This routine would be declared as follows (notice the **`ByRef`** keyword in the third parameter):

```
Declare Function SystemParametersInfo Lib "user" (ByVal action As Integer, _
ByVal uParam As Integer,ByRef pInfo As Integer, ByVal updateINI As Integer) _
As Integer
```

Strings can be passed by reference or by value. When they are passed by reference, a pointer to a pointer to a null-terminated string is passed. When they are passed by value, the compiler passes a pointer to a null-terminated string (i.e., a C string).

When passing a string by reference, the external routine can change the pointer or modify the contents of the existing. If an external routine modifies a passed string variable (regardless of whether the string was passed by reference or by value), then there must be sufficient space within the string to hold the returned characters. This can be accomplished using the **Space** function, as shown in the following example:

```
Declare Sub GetWindowsDirectory Lib "kernel" (ByVal dirname$, ByVal length%)

Sub Main
  Dim s As String
  s = Space(128)
  GetWindowsDirectory s,128
End Sub
```

Another alternative to ensure that a string has sufficient space is to declare the string with a fixed length:

```
Declare Sub GetWindowsDirectory Lib "kernel" (ByVal dirname$, ByVal length%)

Sub Main
  Dim s As String * 128
  GetWindowsDirectory s,len(s)
End Sub
```

Calling conventions with external routines

For external routines, the argument list must exactly match that of the referenced routine. When calling an external subroutine or function, the compiler needs to be told how that routine expects to receive its parameters and who is responsible for cleanup of the stack. The following table describes the macro language’s calling conventions and how these translate to those supported by C.

Macro Call	C Call	Characteristics
StdCall	**_stdcall**	Arguments are pushed right to left. The called function performs stack cleanup. This is the default.
Pascal	**pascal**	Arguments are pushed left to right. The called function performs stack cleanup
Cdecl	**cdecl**	Arguments are pushed right to left. The caller performs stack cleanup.

Passing null pointers

For external routines defined to receive strings by value, the compiler passes uninitialized strings as null pointers (a pointer whose value is 0). The constant **ebNullString** can be used to force a null pointer to be passed as shown below:

```
Declare Sub Foo Lib "sample" (ByVal lpName As Any)

Sub Main
  Foo ebNullString      'pass a null pointer
End Sub
```

Another way to pass a null pointer is to declare the parameter that is to receive the null pointer as type **`Any`**, then pass a long value 0 by value:

```
Declare Sub Foo Lib "sample" (ByVal lpName As Any)

Sub Main
  Foo ByVal 0&        'Pass a null pointer.
End Sub
```

Passing data to external routines

The following table shows how the different data types are passed to external routines:

Data Type	Passed As
`ByRef Boolean`	Pointer to a 2-byte value containing –1 or 0.
`ByVal Boolean`	2-byte value containing –1 or 0.
`ByVal Integer`	Pointer to a 2-byte short integer.
`ByRef Integer`	2-byte short integer.
`ByVal Long`	Pointer to a 4-byte long integer.
`ByRef Long`	4-byte long integer.
`ByRef Single`	Pointer to a 4-byte IEEE floating-point value (a **`float`**).
`ByVal Single`	4-byte IEEE floating-point value (a **`float`**).
`ByRef Double`	Pointer to an 8-byte IEEE floating-point value (a **`double`**).
`ByVal Double`	8-byte IEEE floating-point value (a **`double`**).
`ByVal String`	A pointer to a null-terminated string. With strings containing embedded nulls (**`Chr$(0)`**), it is not possible to determine which null represents the end of the string; therefore, the first null is considered the string terminator. An external routine can freely change the content of a string. It cannot, however, write beyond the end of the null terminator.
`ByRef String`	A pointer to a pointer to a null-terminated string. With strings containing embedded nulls (**`Chr$(0)`**), it is not possible to determine which null represents the end of the string; therefore, the first null is considered the string terminator. An external routine can freely change the content of a string. It cannot, however, write beyond the end of the null terminator.
`ByRef Variant`	A pointer to a 16-byte variant structure. This structure contains a 2-byte type (the same as that returned by the VarType function), followed by 6-bytes of slop (for alignment), followed by 8-bytes containing the value.
`ByVal Variant`	A 16-byte variant structure. This structure contains a 2-byte type (the same as that returned by the VarType function), followed by 6-bytes of slop (for alignment), followed by 8-bytes containing the value.

Data Type	Passed As
ByVal Object	For data objects, a 4-byte unsigned long integer. This value can only be used by external routines written specifically for the macro language. For OLE Automation objects, a 32-bit pointer to an LPDISPATCH handle is passed.
ByRef Object	For data objects, a pointer to a 4-byte unsigned long integer that references the object. This value can only be used by external routines written specifically for the macro language. For OLE Automation objects, a pointer to an LPDISPATCH value is passed.
ByVal User-defined type	The entire structure is passed to the external routine. It is important to remember that structures in the macro language are packed on 2-byte boundaries, meaning that the individual structure members may not be aligned consistently with similar structures declared in C.
ByRef User-defined type	A pointer to the structure. It is important to remember that structures in the macro language are packed on 2-byte boundaries, meaning that the individual structure members may not be aligned consistently with similar structures declared in C.
Arrays	A pointer to a packed array of elements of the given type. Arrays can only be passed by reference.
Dialogs	Dialogs cannot be passed to external routines.

Only variable-length strings can be passed to external routines; fixed-length strings are automatically converted to variable-length strings.

The compiler passes data to external functions consistent with that routine's prototype as defined by the **Declare** statement. There is one exception to this rule: you can override **ByRef** parameters using the **ByVal** keyword when passing individual parameters. The following example shows a number of different ways to pass an **Integer** to an external routine called Foo:

```
Declare Sub Foo Lib "MyLib" (ByRef i As Integer)

Sub Main
  Dim i As Integer
  i = 6
  Foo 6        'Passes a temporary integer (value 6) by
            'reference
  Foo i        'Passes variable "i" by reference
  Foo (i)        'Passes a temporary integer (value 6) by
            'reference
  Foo i + 1        'Passes temporary integer (value 7) by
            'reference
  Foo ByVal i      'Passes i by value
End Sub
```

The above example shows that the only way to override passing a value by reference is to use the **ByVal** keyword.

Note Use caution when using the ByVal keyword in this way. The external routine **`Foo`** expects to receive a pointer to an **`Integer`**—a 32-bit value; using **`ByVal`** causes the compiler to pass the **`Integer`** by value—a 16-bit value. Passing data of the wrong size to any external routine will have unpredictable results.

Returning values from external routines

The compiler supports the following values returned from external routines: **`Integer`**, **`Long`**, **`Single`**, **`Double`**, **`String`**, **`Boolean`**, and all object types. When returning a **`String`**, the compiler assumes that the first null-terminator is the end of the string.

Calling external routines

The compiler makes a copy of all data passed to external routines. This allows other simultaneously executing macros to continue executing before the external routine returns.

Care must be exercised when passing the same by-reference variable twice to external routines. When returning from such calls, the compiler must update the real data from the copies made prior to calling the external function. Since the same variable was passed twice, you will be unable to determine which variable will be updated.

External routines are contained in DLLs. The libraries containing the routines are loaded when the routine is called for the first time (i.e., not when the macro is loaded). This allows a macro to reference external DLLs that potentially do not exist.

Note You cannot execute routines contained in 16-bit Windows DLLs.

All the Windows API routines are contained in DLLs, such as "user32", "kernel32", and "gdi32". The file extension ".exe" is implied if another extension is not given.

The **`Pascal`** and **`StdCall`** calling conventions are identical. Furthermore, the arguments are passed using C ordering regardless of the calling convention—right to left on the stack.

If the **`LibName$`** parameter does not contain an explicit path to the DLL, the following search will be performed for the DLL (in this order):

1. The directory containing the compiler
2. The current directory
3. The Windows system directory
4. The Windows directory
5. All directories listed in the path environment variable

If the first character of **AliasName$** is #, then the remainder of the characters specify the ordinal number of the routine to be called. For example, the following two statements are equivalent (under Win32, **GetCurrentTime** is defined as **GetTickCount**, ordinal 300, in kernel32.dll):

```
Declare Function GetTime Lib "kernel32.dll" Alias "GetTickCount" () As Long

Declare Function GetTime Lib "kernel32.dll" Alias "#300" () As Long
```

Both **name** and **AliasName$** are case-sensitive.

All strings passed by value are converted to MBCS strings. Similarly, any string returned from an external routine is assumed to be a null-terminated MBCS string.

The compiler does not perform an increment on OLE automation objects before passing them to external routines. When returned from an external function, it assumes that the properties and methods of the OLE automation object are UNICODE and that the object uses the default system locale.

Example

```
Declare Function GetModuleHandle& Lib "kernel32" Alias "GetModuleHandleA" (ByVal_
name2 As_ String)

Declare Function GetProfileString& Lib "Kernel32" Alias "GetProfileStringA" (ByVal_
SName As_ String, ByVal KName As String, ByVal Def As String, ByVal Ret As String,_
ByVal Size As Long)

Sub Main
  SName$ = "Intl"      'Win.ini section name.
  KName$ = "sCountry"      'Win.ini country setting.
  ret$ = String$(255, 0)    'Initialize return string.
  If GetProfileString(SName$,KName$,"",ret$,Len(ret$)) Then
    Session.Echo "Your country setting is: " & ret$
  Else
    Session.Echo "There is no country setting in your win.ini file."
  End If
  If GetModuleHandle("Progman") Then
    Session.Echo "Progman is loaded."
  Else
    Session.Echo "Progman is not loaded."
  End If
End Sub
```

See Also Macro Control and Compilation on page 10

DefType

Syntax **{DefInt | DefLng | DefStr | DefSng | DefDbl | DefCur | DefObj | DefVar | DefBool | DefDate} letterrange**

Description Establishes the default type assigned to undeclared or untyped variables. The **DefType** statement controls automatic type declaration of variables. Normally, if a variable is encountered that hasn't yet been declared with the **Dim**, **Public**, or **Private** statement or does not appear with an explicit type-declaration character, then that variable is declared implicitly as a variant (**DefVar** A–Z). This can be changed using the **DefType** statement to specify starting letter ranges for **Type** other than integer. The

letterrange parameter is used to specify starting letters. Thus, any variable that begins with a specified character will be declared using the specified **Type**.

The syntax for **letterrange** is:

```
letter [-letter] [,letter [-letter]]...
```

DefType variable types are superseded by an explicit type declaration using either a type-declaration character or the **Dim**, **Public**, or **Private** statement.

The **DefType** statement only affects how macros are compiled and has no effect at runtime.

The **DefType** statement can only appear outside all **Sub** and **Function** declarations.

The following table describes the data types referenced by the different variations of the **DefType** statement:

Statement	Data Type
DefInt	Integer
DefLng	Long
DefStr	String
DefSng	Single
DefDbl	Double
DefCur	Currency
DefObj	Object
DefVar	Variant
DefBool	Boolean
DefDate	Date

Example

```
DefStr a-l
DefLng m-r
DefSng s-u
DefDbl v-w
DefInt x-z
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  a = 100.52
  m = 100.52
  s = 100.52
  v = 100.52
  x = 100.52
  mesg = "The values are:"
  mesg = mesg & "(String) a: " & a
  mesg = mesg & "(Long) m: " & m
  mesg = mesg & "(Single) s: " & s
  mesg = mesg & "(Double) v: " & v
```

```
  mesg = mesg & "(Integer) x: " & x
  Session.Echo mesg
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Dialog (function)

Syntax **`Dialog(DialogVariable [,[DefaultButton] [,Timeout]])`**

Description Displays the dialog associated with **`DialogVariable`**, returning an **`Integer`** indicating which button was clicked. The **`Dialog`** function returns any of the following values:

Value	Function
–1	The OK button was clicked.
0	The Cancel button was clicked.
>0	A push button was clicked. The returned number represents which button was clicked based on its order in the dialog template (1 is the first push button, 2 is the second push button, and so on).

The **`Dialog`** function accepts the following parameters:

Parameter	Description
`DialogVariable`	Name of a variable that has previously been dimensioned as a user dialog. This is accomplished using the Dim statement: `Dim MyDialog As MyTemplate.` All dialog variables are local to the Sub or Function in which they are defined. Private and public dialog variables are not allowed.
`DefaultButton`	An Integer specifying which button is to act as the default button in the dialog. The value of `DefaultButton` can be any of the following: • `–1` This value indicates that the OK button, if present, should be used as the default. • `0` This value indicates that the Cancel button, if present, should be used as the default. • `>0` This value indicates that the `N`th button should be used as the default. This number is the index of a push button within the dialog template. If `DefaultButton` is not specified, then –1 is used. If the number specified by `DefaultButton` does not correspond to an existing button, then there will be no default button. The default button appears with a thick border and is selected when the user presses Enter on a control other than a push button.
`Timeout`	An integer specifying the number of milliseconds to display the dialog before automatically dismissing it. If `Timeout` is not specified or is equal to 0, then the dialog will be displayed until dismissed by the user. If a dialog has been dismissed due to a timeout, the Dialog function returns 0.

A runtime error is generated if the dialog template specified by `DialogVariable` does not contain at least one of the following statements:

```
PushButton    CancelButton
OKButton      PictureButton
```

Example

```
Sub Main
  Begin Dialog DiskErrorTemplate 16,32,152,48,"Disk Error"
    Text 8,8,100,8,"The disk drive door is open."
    PushButton 8,24,40,14,"Abort",.Abort
    PushButton 56,24,40,14,"Retry",.Retry
    PushButton 104,24,40,14,"Ignore",.Ignore
  End Dialog
  Dim DiskError As DiskErrorTemplate
  r% = Dialog(DiskError,3,0)
  Session.Echo "You selected button: " & r%
End Sub
```

See Also User Interaction on page 16

Dialog (statement)

Syntax

```
Dialog DialogVariable [,[DefaultButton] [,Timeout]]
```

Description Same as the **Dialog** function, except that the **Dialog** statement does not return a value. (See **Dialog** [function].)

Example

```
Sub Main
  Begin Dialog DiskErrorTemplate 16,32,152,48,"Disk Error"
    Text 8,8,100,8,"The disk drive door is open."
    PushButton 8,24,40,14,"Abort",.Abort
    PushButton 56,24,40,14,"Retry",.Retry
    PushButton 104,24,40,14,"Ignore",.Ignore
  End Dialog
  Dim DiskError As DiskErrorTemplate
  Dialog DiskError,3,0
End Sub
```

See Also User Interaction on page 16

Dialogs (topic)

The compiler displays all runtime dialogs in the following fonts:

- 8-point MS Sans Serif font for non-MBCS systems
- The default system font for MBCS systems

The default help key is F1.

See Also User Interaction on page 16

Dim

Syntax

```
Dim name [(<submacros>)] [As [New] type] [,name [(<submacros>)] [As [New] type]]...
```

Description Declares a list of local variables and their corresponding types and sizes. If a type-declaration character is used when specifying **name** (such as %, @, &, $, or !), the optional [**As type**] expression is not allowed. For example, the following are allowed:

```
Dim Temperature As Integer
Dim Temperature%
```

The **submacros** parameter allows the declaration of dynamic and fixed arrays. The **submacros** parameter uses the following syntax:

```
[lower to] upper [,[lower to] upper]...
```

The **lower** and **upper** parameters are integers specifying the lower and upper bounds of the array. If **lower** is not specified, then the lower bound as specified by **Option Base** is used (or 1 if no **Option Base** statement has been encountered). You can have a maximum of 60 array dimensions.

The total size of an array (not counting space for strings) is limited to 64K. Dynamic arrays are declared by not specifying any bounds:

```
Dim a()
```

The **type** parameter specifies the type of the data item being declared. It can be any of the following data types: **String**, **Integer**, **Long**, **Single**, **Double**, **Currency**, **Object**, data object, built-in data type, or any user-defined data type. When specifying explicit object types, you can use the following syntax for **type**:

```
module.class
```

where **module** is the name of the module in which the object is defined and **class** is the type of object. For example, to specify the OLE automation variable for Excel's Application object, you could use the following code:

```
Dim a As Excel.Application
```

Note Explicit object types can only be specified for data objects and early bound OLE automation objects—i.e., objects whose type libraries have been registered with the compiler.

A **Dim** statement within a subroutine or function declares variables local to that subroutine or function. If the **Dim** statement appears outside of any subroutine or function declaration, then that variable has the same scope as variables declared with the **Private** statement.

Fixed-length strings

Fixed-length strings are declared by adding a length to the **String** type-declaration character:

```
Dim name As String * length
```

where **length** is a literal number specifying the string's length.

Implicit variable declaration

If the compiler encounters a variable that has not been explicitly declared with **Dim**, then the variable will be implicitly declared using the specified type-declaration character (#, %, @, $, or &). If the variable appears without a type-declaration character, then the first letter is matched against any pending **DefType** statements, using the specified type if found. If no **DefType** statement has been encountered corresponding to the first letter of the variable name, then **Variant** is used.

Declaring explicit OLE automation objects

The **`Dim`** statement can be used to declare variables of an explicit object type for objects known to the compiler through type libraries. This is accomplished using the following syntax:

```
Dim name As application.class
```

The **`application`** parameter specifies the application used to register the OLE automation object and **`class`** specifies the specific object type as defined in the type library. Objects declared in this manner are early bound, meaning that the compiler is able to resolve method and property information at compile time, improving the performance when invoking methods and properties of that object variable.

Creating new objects

The optional **`New`** keyword is used to declare a new instance of the specified data object. This keyword cannot be used when declaring arrays or OLE automation objects.

At runtime, the application or extension that defines that object type is notified that a new object is being defined. The application responds by creating a new physical object (within the appropriate context) and returning a reference to that object, which is immediately assigned to the variable being declared.

When that variable goes out of scope (i.e., the **`Sub`** or **`Function`** procedure in which the variable is declared ends), the application is notified. The application then performs some appropriate action, such as destroying the physical object.

Initial values

All declared variables are given initial values, as described in the following table:

Data Type	Initial Value
Integer	0
Long	0
Double	0.0
Single	0.0
Date	December 30, 1899 00:00:00
Currency	0.0
Boolean	False
Object	Nothing
Variant	Empty

Data Type	Initial Value
String	"" (zero-length string)
User-defined type	Each element of the structure gets an initial value as described above.
Arrays	Each element of the array gets an initial value as described above.

Naming conventions

Variable names must follow these naming rules:

- Must start with a letter.
- May contain letters, digits, and the underscore character (_); punctuation is not allowed. The exclamation point (!) can appear within the name as long as it is not the last character, in which case it is interpreted as a type-declaration character.
- The last character of the name can be any of the following type-declaration characters: #, @, %, !, &, and $.
- Must not exceed 80 characters in length.
- Cannot be a reserved word.

Examples The following examples use the Dim statement to declare various variable types.

```
Sub Main
    Dim i As Integer
    Dim l&                          'Long
    Dim s As Single
    Dim d#                          'Double
    Dim c$                          'String
    Dim MyArray(10) As Integer      '10 element integer array
    Dim MyStrings$(2,10)            '2-10 element string arrays
    Dim Filenames$(5 to 10)         '6 element string array
    Dim Values(1 to 10, 100 to 200) '111 element variant array
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Dir, Dir$

Syntax `Dir[$] [(pathname [,attributes])]`

Description Returns a **`String`** containing the first or next file matching **`pathname`**. If **`pathname`** is specified, then the first file matching that **`pathname`** is returned. If **`pathname`** is not specified, then the next file matching the initial **`pathname`** is returned.

`Dir$` returns a **`String`**, whereas **`Dir`** returns a **`String`** variant.

The **`Dir$/Dir`** functions take the following named parameters:

Parameter	Description
`pathname`	String containing a file specification. If this parameter is specified, then **`Dir$`** returns the first file matching this file specification. If this parameter is omitted, then the next file matching the initial file specification is returned. If no path is specified in **`pathname`**, then all files are returned from the current directory.
`attributes`	Integer specifying attributes of files you want included in the list, as described below. If this parameter is omitted, then only the normal, read-only, and archive files are returned.

An error is generated if **`Dir$`** is called without first calling it with a valid **`pathname`**.

If there is no matching **`pathname`**, then a zero-length string is returned.

Wildcards

The **`pathname`** argument can include wildcards, such as * and ?. The * character matches any sequence of zero or more characters, whereas the ? character matches any single character. Multiple *s and ?s can appear within the expression to form complete searching patterns. The following table shows some examples:

This Pattern	Matches These Files	Not TheseFiles
S.TXT	SAMPLE.TXT, GOOSE.TXT, SAMS.TXT	SAMPLE, SAMPLE.DAT
C*T.TXT	CAT.TXT	CAP.TXT, ACATS.TXT
C*T	CAT, CAP.TXT	CAT.DOC
C?T	CAT, CUT	CAT.TXT, CAPITCT
*	(All files)	

Attributes

You can control which files are included in the search by specifying the optional attributes parameter. The **`Dir`**, **`Dir$`** functions always return all normal, read-only, and archive files (**`ebNormal Or ebReadOnly Or ebArchive`**). To include additional files, you can specify any combination of the following attributes (combined with the **`Or`** operator):

Constant	Value	Includes
`ebNormal`	0	Read-only, archive, subdir, and none
`ebHidden`	2	Hidden files
`ebSystem`	4	System files
`ebVolume`	8	Volume label
`ebDirectory`	16	Subdirectories

Example

```
Const crlf = Chr$(13) + Chr$(10)
```

```
Sub Main
  Dim a$(10)
  a(1) = Dir$("*.*")
  i% = 1
  While (a(i%) <> "") And (i% < 10)
    i% = i% + 1
    a(i%) = Dir$
  Wend
  Session.Echo a(1) & crlf & a(2) & crlf & a(3) & crlf & a(4)
End Sub
```

See Also Drive, Folder, and File Access on page 4

DiskDrives

Syntax `DiskDrives array()`

Description Fills the specified **String** or **Variant** array with a list of valid drive letters. The **array**() parameter specifies either a zero- or a one-dimensioned array of strings or variants. The array can be either dynamic or fixed.

If **array**() is dynamic, then it will be redimensioned to exactly hold the new number of elements. If there are no elements, then the array will be redimensioned to contain no dimensions. You can use the **LBound**, **UBound**, and **ArrayDims** functions to determine the number and size of the new array's dimensions.

If the array is fixed, each array element is first erased, then the new elements are placed into the array. If there are fewer elements than will fit in the array, then the remaining elements are initialized to zero-length strings (for **String** arrays) or **Empty** (for **Variant** arrays). A runtime error results if the array is too small to hold the new elements.

Example

```
Sub Main
  Dim drive$()
  DiskDrives drive$
  Session.Echo "Available Disk Drives:<CR><LF>"
  For i= 0 to UBound(drive$)
    Session.Echo drive$ & "<CR><LF>"
  Next i
End Sub
```

See Also Drive, Folder, and File Access on page 4

DiskFree

Syntax `DiskFree&([drive$])`

Description Returns a **Long** containing the free space (in bytes) available on the specified drive. If **drive$** is zero-length or not specified, then the current drive is assumed. Only the first character of the **drive$** string is used.

Example

```
Sub Main
  s$ = "c"
  i# = DiskFree(s$)
  Session.Echo "Free disk space on drive '" & s$ & "' is: " & i#
End Sub
```

See Also Drive, Folder, and File Access on page 4

DlgCaption (function)

Syntax `DlgCaption[()]`

Description Returns a string containing the caption of the active user-defined dialog. This function returns a zero-length string if the active dialog has no caption.

See Also User Interaction on page 16

DlgCaption (statement)

Syntax `DlgCaption text`

Description Changes the caption of the current dialog to **`text`**.

Example

```
Function DlgProc(c As String,a As Integer,v As Integer)
  If a = 1 Then
    DlgCaption choose(DlgValue("OptionGroup1") + 1, _
      "Blue","Green")
  ElseIf a = 2 Then
    DlgCaption choose(DlgValue("OptionGroup1") + 1, _
      "Blue","Green")
  End If
End Function

Sub Main
  Begin Dialog UserDialog ,,149,45,"Untitled",.DlgProc
    OKButton 96,8,40,14
    OptionGroup .OptionGroup1
      OptionButton 12,12,56,8,"Blue",.OptionButton1
      OptionButton 12,28,56,8,"Green",.OptionButton2
  End Dialog
  Dim d As UserDialog
  Dialog d
End Sub
```

See Also User Interaction on page 16

DlgControlId

Syntax `DlgControlId(ControlName$)`

Description Returns an **`Integer`** containing the index of the specified control as it appears in the dialog template. The first control in the dialog template is at index 0, the second is at index 1, and so on. The

ControlName$ parameter contains the name of the **.Identifier** parameter associated with that control in the dialog template.

The macro statements and functions that dynamically manipulate dialog controls identify individual controls using either the **.Identifier** name of the control or the control's index. Using the index to refer to a control is slightly faster but results in code that is more difficult to maintain.

Example

```
Function DlgProc(ControlName$,Action%,SuppValue%) As Integer
  'If a control is clicked, disable the next three controls.

  If Action% = 2 Then
    'Enable the next three controls.
    start% = DlgControlId(ControlName$)
    For i = start% + 1 To start% + 3
      DlgEnable i,True
    Next i
    DlgProc = 1    'Don't close the dialog.
  End If
End Function
```

See Also User Interaction on page 16

DlgEnable (function)

Syntax **DlgEnable(ControlName$ | ControlIndex)**

Description Returns **True** if the specified control is enabled; returns **False** otherwise. Disabled controls are dimmed and cannot receive keyboard or mouse input.

The **ControlName$** parameter contains the name of the **.Identifier** parameter associated with a control in the dialog template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the **ControlIndex** parameter, a control can be referred to using its index in the dialog template (0 is the first control in the template, 1 is the second, and so on).

Note When **ControlIndex** is specified, **OptionGroup** statements do not count as a control.

If you attempt to disable the control with the focus, the compiler will automatically set the focus to the next control in the tab order.

Example

```
If DlgEnable("SaveOptions") Then
  Session.Echo "The Save Options are enabled."
End If
If DlgEnable(10) And DlgVisible(12) Then code = 1 Else code = 2
```

See Also User Interaction on page 16

DlgEnable (statement)

Syntax **`DlgEnable {ControlName$ | ControlIndex} [,isOn]`**

Description Enables or disables the specified control. Disabled controls are dimmed and cannot receive keyboard or mouse input.

The **`isOn`** parameter is an **`Integer`** specifying the new state of the control. It can be any of the following values:

Value	Description
0	The control is disabled.
1	The control is enabled.
Omitted	Toggles the control between enabled and disabled.

Option buttons can be manipulated individually (by specifying an individual option button) or as a group (by specifying the name of the option group).

The **`ControlName$`** parameter contains the name of the **`.Identifier`** parameter associated with a control in the dialog template. Alternatively, by specifying the **`ControlIndex`** parameter, a control can be referred to using its index in the dialog template (0 is the first control in the template, 1 is the second, and so on).

Note When **`ControlIndex`** is specified, **`OptionGroup`** statements do not count as a control.

Example

```
DlgEnable "SaveOptions", False  'Disable the Save Options control.
DlgEnable "EditingOptions"'Toggle a group of option buttons.
For i = 0 To 5
  DlgEnable i,True    'Enable six controls.
Next i
```

See Also User Interaction on page 16

DlgFocus (function)

Syntax **`DlgFocus$[()]`**

Description Returns a **`String`** containing the name of the control with the focus. The name of the control is the **`.Identifier`** parameter associated with the control in the dialog template.

Example

```
If DlgFocus$ = "Files" Then    'Does it have the focus?
  DlgFocus "OK"      'Change the focus to another control.
End If
DlgEnable "Files", False    'Now we can disable the control.
```

See Also User Interaction on page 16

DlgFocus (statement)

Syntax **`DlgFocus ControlName$ | ControlIndex`**

Description Sets focus to the specified control. A runtime error results if the specified control is hidden, disabled, or nonexistent.

The **`ControlName$`** parameter contains the name of the **`.Identifier`** parameter associated with a control in the dialog template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the **`ControlIndex`** parameter, a control can be referred to using its index in the dialog template (0 is the first control in the template, 1 is the second, and so on).

Note When **`ControlIndex`** is specified, **`OptionGroup`** statements do not count as a control.

Example

```
If DlgFocus$ = "Files" Then  'Does it have the focus?
  DlgFocus "OK"    'Change the focus to another control.
End If
DlgEnable "Files", False  'Now we can disable the control.
```

See Also User Interaction on page 16

DlgListBoxArray (function)

Syntax **`DlgListBoxArray({ControlName$ | ControlIndex}, ArrayVariable)`**

Description Fills a listbox, combo box, or drop listbox with the elements of an array, returning an **`Integer`** containing the number of elements that were actually set into the control.

The **`ControlName$`** parameter contains the name of the **`.Identifier`** parameter associated with a control in the dialog template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the **`ControlIndex`** parameter, a control can be referred to using its index in the dialog template (0 is the first control in the template, 1 is the second, and so on).

Note When **`ControlIndex`** is specified, **`OptionGroup`** statements do not count as a control.

The **`ArrayVariable`** parameter specifies a single-dimensioned array used to initialize the elements of the control. If this array has no dimensions, then the control will be initialized with no elements. A runtime error results if the specified array contains more than one dimension. **`ArrayVariable`** can specify an array of any fundamental data type (structures are not allowed). **`Null`** and **`Empty`** values are treated as zero-length strings.

Example

```
Function DlgProc(ControlName$,Action%,SuppValue%) As Integer
  If Action% = 2 And ControlName$ = "Files" Then
    Dim NewFiles$()      'Create a new dynamic array.
    FileList NewFiles$,"*.txt"  'Fill the array with files.
    r% = DlgListBoxArray "Files",NewFiles$
```

```
                 'Set items in the listbox.
    DlgValue "Files",0      'Set the selection to first item.
    DlgProc = 1        'Don't close the dialog.
  End If
  Session.Echo r% & " items were added to the listbox."
End Function
```

See Also User Interaction on page 16

DlgListBoxArray (statement)

Syntax **`DlgListBoxArray {ControlName$ | ControlIndex}, ArrayVariable`**

Description Fills a listbox, combo box, or drop listbox with the elements of an array.

The **`ControlName$`** parameter contains the name of the **`.Identifier`** parameter associated with a control in the dialog template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the **`ControlIndex`** parameter, a control can be referred to using its index in the dialog template (0 is the first control in the template, 1 is the second, and so on).

Note When **`ControlIndex`** is specified, **`OptionGroup`** statements do not count as a control.

The **`ArrayVariable`** parameter specifies a single-dimensioned array used to initialize the elements of the control. If this array has no dimensions, then the control will be initialized with no elements. A runtime error results if the specified array contains more than one dimension. **`ArrayVariable`** can specify an array of any fundamental data type (structures are not allowed). **`Null`** and **`Empty`** values are treated as zero-length strings.

Example

```
Function DlgProc(ControlName$,Action%,SuppValue%) As Integer
  If Action% = 2 And ControlName$ = "Files" Then
    Dim NewFiles$()        'Create a new
                  'dynamic array.
    FileList NewFiles$,"*.txt"    'Fill the array with files.
    DlgListBoxArray "Files",NewFiles$  'Set items in the listbox.
    DlgValue "Files",0        'Set the selection
                  'to the first item.
  End If
End Function
```

See Also User Interaction on page 16

DlgProc

Syntax **`Function DlgProc(ControlName$, Action, SuppValue) As Integer`**

Description Describes the syntax, parameters, and return value for dialog functions. Dialog functions are called by the compiler during the processing of a custom dialog. The name of a dialog function (**`DlgProc`**) appears in the **`Begin Dialog`** statement as the **`.DlgProc`** parameter. Dialog functions require the following parameters:

Parameter	Description
`ControlName$`	String containing the name of the control associated with **`Action`**.
`Action`	Integer containing the action that called the dialog function.
`SuppValue`	Integer of extra information associated with **`Action`**. For some actions, this parameter is not used.

When the compiler displays a custom dialog, the user may click buttons, type text into edit fields, select items from lists, and perform other actions. When these actions occur, the compiler calls the dialog function, passing it the action, the name of the control on which the action occurred, and any other relevant information associated with the action.

The following table describes the different actions sent to dialog functions:

Action	Description
1	This action is sent immediately before the dialog is shown for the first time. This gives the dialog function a chance to prepare the dialog for use. When this action is sent, **`ControlName$`** contains a zero-length string, and **`SuppValue`** is 0.The return value from the dialog function is ignored in this case. Before Showing the dialog: After action 1 is sent, the compiler performs additional processing before the dialog is shown. Specifically, it cycles though the dialog controls checking for visible picture or picture button controls. For each visible picture or picture button control, the compiler attempts to load the associated picture. In addition to checking picture or picture button controls, the compiler automatically hides any control outside the confines of the visible portion of the dialog. This prevents the user from tabbing to controls that cannot be seen. However, it does not prevent you from showing these controls with the **`DlgVisible`** statement in the dialog function.
2	This action is sent when: A button is clicked, such as OK, Cancel, or a push button. In this case, **`ControlName$`** contains the name of the button. **`SuppValue`** contains 1 if an OK button was clicked and 2 if a Cancel button was clicked; **`SuppValue`** is undefined otherwise. If the dialog function returns 0 in response to this action, then the dialog will be closed. Any other value causes the compiler to continue dialog processing. A checkbox's state has been modified. In this case, **`ControlName$`** contains the name of the checkbox, and **`SuppValue`** contains the new state of the checkbox (1 if on, 0 if off). An option button is selected. In this case, **`ControlName$`** contains the name of the option button that was clicked, and **`SuppValue`** contains the index of the option button within the option button group (0-based). The current selection is changed in a listbox, drop listbox, or combo box. In this case, **`ControlName$`** contains the name of the listbox, combo box, or drop listbox, and **`SuppValue`** contains the index of the new item (0 is the first item, 1 is the second, and so on).

Action	Description
3	This action is sent when the content of a text box or combo box has been changed. This action is only sent when the control loses focus. When this action is sent, **`ControlName$`** contains the name of the text box or combo box, and **`SuppValue`** contains the length of the new content. The dialog function's return value is ignored with this action.
4	This action is sent when a control gains the focus. When this action is sent, **`Control-Name$`** contains the name of the control gaining the focus, and **`SuppValue`** contains the index of the control that lost the focus (0-based).The dialog function's return value is ignored with this action.
5	This action is sent continuously when the dialog is idle. If the dialog function returns 1 in response to this action, then the idle action will continue to be sent. If the dialog function returns 0, then the compiler will not send any additional idle actions. When the idle action is sent, **`ControlName$`** contains a zero-length string, and **`SuppValue`** contains the number of times the idle action has been sent so far.
6	This action is sent when the dialog is moved. The **`ControlName$`** parameter contains a zero-length string, and **`SuppValue`** is 0.The dialog function's return value is ignored with this action.

User-defined dialoges cannot be nested. In other words, the dialog function of one dialog cannot create another user-defined dialog. You can, however, invoke any built-in dialog, such as **`Session.Echo`** or **`InputBox$.`**

Within dialog functions, you can use the following additional statements and functions. These statements allow you to manipulate the dialog controls dynamically.

`DlgVisible`	**`DlgText$`**	**`DlgText`**
`DlgSetPicture`	**`DlgListBoxArray`**	**`DlgFocus`**
`DlgEnable`	**`DlgControlId`**	

The dialog function can optionally be declared to return a **`Variant`**. When returning a variable, the compiler will attempt to convert the variant to an **`Integer`**. If the returned variant cannot be converted to an **`Integer`**, then 0 is assumed to be returned from the dialog function.

Example

```
Function SampleDlgProc(ControlName$, Action%, SuppValue%)
  If Action% = 2 And ControlName$ = "Printing" Then
    DlgEnable "PrintOptions",SuppValue%
    SampleDlgProc = 1  'Don't close the dialog.
  End If
End Function

Sub Main
  Begin Dialog SampleDialogTemplate 34, 39, 106, 45, "Sample", _
.SampleDlgProc
    OKButton 4,4,40,14
    CancelButton 4,24,40,14
    CheckBox 56,8,38,8,"Printing",.Printing
```

```
    OptionGroup .PrintOptions
      OptionButton 56,20,51,8,"Landscape",.Landscape
      OptionButton 56,32,40,8,"Portrait",.Portrait
  End Dialog
  Dim SampleDialog As SampleDialogTemplate
  SampleDialog.Printing = 1
  r% = Dialog(SampleDialog)
End Sub
```

See Also User Interaction on page 16

DlgSetPicture

Syntax `DlgSetPicture {ControlName$ | ControlIndex},PictureName$,PictureType`

Description Changes the content of the specified picture or picture button control. The **`DlgSetPicture`** statement accepts the following parameters:

Parameter	Description
`ControlName$`	String containing the name of the **`.Identifier`** parameter associated with a control in the dialog template. A case-insensitive comparison is used to locate the specified control within the template. Alternatively, by specifying the **`ControlIndex`** parameter, a control can be referred to using its index in the dialog template (0 is the first control in the template, 1 is the second, and so on). When **`ControlIndex`** is specified, OptionGroup statements do not count as a control.
`PictureName$`	String containing the name of the picture. If **`PictureType`** is 0, then this parameter specifies the name of the file containing the image. If **`PictureType`** is 10, then **`PictureName$`** specifies the name of the image within the resource of the picture library. If **`PictureName$`** is empty, then the current picture associated with the specified control will be deleted. Thus, a technique for conserving memory and resources would involve setting the picture to empty before hiding a picture control.
`PictureType`	Integer specifying the source for the image. The following sources are supported:
0	The image is contained in a file on disk.
10	The image is contained in the picture library specified by the Begin Dialog statement. When this type is used, the **`PictureName$`** parameter must be specified with the Begin Dialog statement.

Picture controls can contain either bitmaps or WMFs (Windows metafiles). When extracting images from a picture library, the compiler assumes that the resource type for metafiles is 256.

Picture libraries are implemented as DLLs.

Examples

```
'Set picture from a file.
DlgSetPicture "Picture1","\windows\checks.bmp",0
'Set control 10's image from a library.
DlgSetPicture 27,"FaxReport",10
```

See Also User Interaction on page 16

DlgText

Syntax `DlgText {ControlName$ | ControlIndex}, NewText$`

Description Changes the text content of the specified control. The effect of this statement depends on the type of the specified control:

Control Type	Effect of DlgText
Picture	Runtime error.
Option group	Runtime error.
Drop listbox	If an exact match cannot be found, the **DlgText** statement searches from the first item looking for an item that starts with **NewText$**. If no match is found, then the selection is removed.
OK button	Sets the label of the control to **NewText$**.
Cancel button	Sets the label of the control to **NewText$**.
Push button	Sets the label of the control to **NewText$**.
Listbox	Sets the current selection to the item matching **NewText$**. If an exact match cannot be found, the **DlgText** statement searches from the first item looking for an item that starts with **NewText$**. If no match is found, then the selection is removed.
Combo box	Sets the content of the edit field of the combo box to **NewText$**.
Text	Sets the label of the control to **NewText$**.
Text box	Sets the content of the text box to **NewText$**.
Group box	Sets the label of the control to **NewText$**.
Option button	Sets the label of the control to **NewText$**.

The ControlName$ parameter contains the name of the **.Identifier** parameter associated with a control in the dialog template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the **ControlIndex** parameter, a control can be referred to using its index in the dialog template (0 is the first control in the template, 1 is the second, and so on).

Note When **ControlIndex** is specified, **OptionGroup** statements do not count as a control.

Example

```
DlgText "GroupBox1","Save Options" 'Change text of group box 1.
If DlgText$(9) = "Save Options" Then
  DlgText 9,"Editing Options"'Change text to "Editing Options".
End If
```

See Also User Interaction on page 16

DlgText$

Syntax `DlgText$(ControlName$ | ControlIndex)`

Description Returns the text content of the specified control. The text returned depends on the type of the specified control:

Control Type	Value Returned by DlgText$
Picture	No value is returned. A runtime error occurs.
Option group	No value is returned. A runtime error occurs.
Drop listbox	Returns the currently selected item. A zero-length string is returned if no item is currently selected.
OK button	Returns the label of the control.
Cancel button	Returns the label of the control.
Push button	Returns the label of the control.
Listbox	Returns the currently selected item. A zero-length string is returned if no item is currently selected.
Combo box	Returns the content of the edit field portion of the combo box.
Text	Returns the label of the control.
Text box	Returns the content of the control.
Group box	Returns the label of the control.
Option button	Returns the label of the control.

`The ControlName$` parameter contains the name of the **`.Identifier`** parameter associated with a control in the dialog template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the **`ControlIndex`** parameter, a control can be referred to using its index in the dialog template (0 is the first control in the template, 1 is the second, and so on).

Note When **`ControlIndex`** is specified, **`OptionGroup`** statements do not count as a control.

Example

```
Session.Echo DlgText$(10)        'Display the text in the tenth control.
If DlgText$("SaveOptions") = "EditingOptions" Then
  Session.Echo "You are currently viewing the editing options."
End If
```

See Also User Interaction on page 16

DlgValue (function)

Syntax `DlgValue(ControlName$ | ControlIndex)`

Description Returns an **`Integer`** indicating the value of the specified control. The value of any given control depends on its type, according to the following table:

Control Type	DlgValue Returns
Option group	The index of the selected option button within the group (0 is the first option button, 1 is the second, and so on).
Listbox	The index of the selected item.
Drop listbox	The index of the selected item.
Checkbox	1 if the checkbox is checked; 0 otherwise.

A runtime error is generated if **`DlgValue`** is used with controls other than those listed in the above table.

The **`ControlName$`** parameter contains the name of the **`.Identifier`** parameter associated with a control in the dialog template. Alternatively, by specifying the **`ControlIndex`** parameter, a control can be referred to using its index in the dialog template (0 is the first control in the template, 1 is the second, and so on).

Note When **`ControlIndex`** is specified, **`OptionGroup`** statements do not count as a control.

Example See **`DlgValue`** (statement).

See Also User Interaction on page 16

DlgValue (statement)

Syntax `DlgValue {ControlName$ | ControlIndex},Value`

Description Changes the value of the given control. The value of any given control is an **`Integer`** and depends on its type, according to the following table:

Control Type	Description of Value
Option group	The index of the new selected option button within the group (0 is the first option button, 1 is the second, and so on).
Listbox	The index of the new selected item.
Drop listbox	The index of the new selected item.
Checkbox	1 if the checkbox is to be checked; 0 to remove the check.

A runtime error is generated if DlgValue is used with controls other than those listed in the above table.

The **ControlName$** parameter contains the name of the **.Identifier** parameter associated with a control in the dialog template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the **ControlIndex** parameter, a control can be referred to using its index in the dialog template (0 is the first control in the template, 1 is the second, and so on).

Note When **ControlIndex** is specified, **OptionGroup** statements do not count as a control.

Example

```
If DlgValue("MyCheckBox") = 1 Then
  DlgValue "MyCheckBox",0
Else
  DlgValue "MyCheckBox",1
End If
```

See Also User Interaction on page 16

DlgVisible (function)

Syntax **DlgVisible(ControlName$ | ControlIndex)**

Description Returns **True** if the specified control is visible; returns **False** otherwise.

The **ControlName$** parameter contains the name of the **.Identifier** parameter associated with a control in the dialog template. Alternatively, by specifying the **ControlIndex** parameter, a control can be referred to using its index in the template (0 is the first control in the template, 1 is the second, and so on).

Note When **ControlIndex** is specified, **OptionGroup** statements do not count as a control.

A runtime error is generated if **DlgVisible** is called when no user dialog is active.

Example

```
If DlgVisible("Portrait") Then Beep
If DlgVisible(10) And DlgVisible(12) Then
  Session.Echo "The 10th and 12th controls are visible."
End If
```

See Also User Interaction on page 16

DlgVisible (statement)

Syntax **DlgVisible {ControlName$ | ControlIndex} [,isOn]**

Description Hides or shows the specified control. Hidden controls cannot be seen in the dialog and cannot receive the focus using Tab.

The **isOn** parameter is an **Integer** specifying the new state of the control. It can be any of the following values:

Value	Description
1	The control is shown.
0	The control is hidden.
Omitted	Toggles the visibility of the control.

Option buttons can be manipulated individually (by specifying an individual option button) or as a group (by specifying the name of the option group).

The **ControlName$** parameter contains the name of the **.Identifier** parameter associated with a control in the dialog template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the **ControlIndex** parameter, a control can be referred to using its index in the dialog template (0 is the first control in the template, 1 is the second, and so on).

Note When **ControlIndex** is specified, **OptionGroup** statements do not count as a control.

Picture Caching

When the dialog is first created and before it is shown, the compiler calls the dialog function with **action** set to 1. At this time, no pictures have been loaded into the picture controls contained in the dialog template. After control returns from the dialog function and before the dialog is shown, the compiler will load the pictures of all visible picture controls. Thus, it is possible for the dialog function to hide certain picture controls, which prevents the associated pictures from being loaded and causes the dialog to load faster. When a picture control is made visible for the first time, the associated picture will then be loaded.

Example

```
Sub EnableGroup(start%, finish%)
  For i = 6 To 13          'Disable all options.
    DlgVisible i, False
  Next i
  For i = start% To finish%      'Enable only the right ones.
    DlgVisible i, True
  Next i
End Sub

Function DlgProc(ControlName$, Action%, SuppValue%)
  If Action% = 1 Then
    DlgValue "WhichOptions",0    'Set to save options.
    EnableGroup 6, 8        'Enable the save options.
  End If
  If Action% = 2 And ControlName$ = "SaveOptions" Then
    EnableGroup 6, 8        'Enable the save options.
    DlgProc = 1          'Don't close the dialog.
  End If
  If Action% = 2 And ControlName$ = "EditingOptions" Then
    EnableGroup 9, 13        'Enable the editing options.
```

```
    DlgProc = 1          'Don't close the dialog.
  End If
End Function

Sub Main
  Begin Dialog OptionsTemplate 33, 33, 171, 134, "Options", .DlgProc
    'Background (controls 0-5)
    GroupBox 8, 40, 152, 84, ""
    OptionGroup .WhichOptions
      OptionButton 8, 8, 59, 8, "Save Options",.SaveOptions
      OptionButton 8, 20, 65, 8, "Editing Options",.EditingOptions
    OKButton 116, 7, 44, 14
    CancelButton 116, 24, 44, 14
    'Save options (controls 6-8)
    CheckBox 20, 56, 88, 8, "Always create backup",.CheckBox1
    CheckBox 20, 68, 65, 8, "Automatic save",.CheckBox2
    CheckBox 20, 80, 70, 8, "Allow overwriting",.CheckBox3
    'Editing options (controls 9-13)
    CheckBox 20, 56, 65, 8, "Overtype mode",.OvertypeMode
    CheckBox 20, 68, 69, 8, "Uppercase only",.UppercaseOnly
    CheckBox 20, 80, 105, 8, "Automatically check syntax",.AutoCheckSyntax
    CheckBox 20, 92, 73, 8, "Full line selection",.FullLineSelection
    CheckBox 20, 104, 102, 8, "Typing replaces selection",.TypingReplacesText
  End Dialog
  Dim OptionsDialog As OptionsTemplate
  Dialog OptionsDialog
End Sub
```

See Also User Interaction on page 16

Do...Loop

Syntax 1 **`Do {While | Until} condition statements Loop`**

Syntax 2

```
Do
  statements
Loop {While | Until} condition
```

Syntax 3

```
Do
  statements
Loop
```

Description Repeats a block of statements while a condition is **`True`** or until a condition is **`True`**. If the {**`While`** | **`Until`**} conditional clause is not specified, then the loop repeats the statements forever (or until the compiler encounters an **`Exit Do`** statement).

The **`condition`** parameter specifies any **`Boolean`** expression.

Due to errors in program logic, you can inadvertently create infinite loops in your code. When you're running a macro within the macro editor, you can break out of an infinite loop by pressing Ctrl+Break.

Examples This first example uses the Do...While statement, which performs the iteration, then checks the condition, and repeats if the condition is True.

```
Sub Main
  Dim a$(100)
  i% = -1
  Do
    i% = i% + 1
    If i% = 0 Then
      a(i%) = Dir$("*")
    Else
      a(i%) = Dir$
    End If
  Loop While (a(i%) <> "" And i% <= 99)
  Session.Echo str$(i%) & " files found" & "<CR><LF>
```

This second example uses the **`Do While`**...Loop, which checks the condition and then repeats if the condition is True.

```
  Dim a$(100)
  i% = 0
  a(i%) = Dir$("*")
  Do While a(i%) <> "" And i% <= 99
    i% = i% + 1
    a(i%) = Dir$
  Loop
  Session.Echo str$(i%) & " files found" & "<CR><LF>
```

This third example uses the **`Do Until`**...Loop, which does the iteration and then checks the condition and repeats if the condition is True.

```
  Dim a$(100)
  i% = 0
  a(i%) = Dir$("*")
  Do Until a(i%) = "" Or i% = 100
    i% = i% + 1
    a(i%) = Dir$
  Loop
  Session.Echo str$(i%) & " files found" & "<CR><LF>
```

This last example uses the **`Do...Until`** Loop, which performs the iteration first, checks the condition, and repeats if the condition is True.

```
  Dim a$(100)
  i% = -1
  Do
    i% = i% + 1
    If i% = 0 Then
      a(i%) = Dir$("*")
    Else
      a(i%) = Dir$
    End If
  Loop Until (a(i%) = "" Or i% = 100)
  Session.Echo str$(i%) & " files found" & "<CR><LF>
End Sub
```

See Also Macro Control and Compilation on page 10

DoEvents (function)

Syntax **`DoEvents[()]`**

Description Yields control to other applications, returning an **`Integer`** 0. This statement yields control to the operating system, allowing other applications to process mouse, keyboard, and other messages.

If a **`SendKeys`** statement is active, this statement waits until all the keys in the queue have been processed.

Example See DoEvents (statement).

See Also Operating System Control on page 15

DoEvents (statement)

Syntax **`DoEvents`**

Description Yields control to other applications. This statement yields control to the operating system, allowing other applications to process mouse, keyboard, and other messages.

If a **`SendKeys`** statement is active, this statement waits until all the keys in the queue have been processed.

Examples This first example shows a macro that takes a long time and hogs the system. The subroutine explicitly yields to allow other applications to execute.

```
Sub Main
  Open "test.txt" For Output As #1
  For i = 1 To 10000
    Print #1,"This is a test of the system and stuff."
    DoEvents
  Next i
  Close #1
End Sub
```

In this second example, the DoEvents statement is used to wait until the queue has been completely flushed.

```
Sub Main
  AppActivate "Notepad"      'Activate Notepad.
  SendKeys "This is a test.",False  'Send some keys.
  DoEvents            'Wait for the keys to play back.
End Sub
```

See Also Operating System Control on page 15

Double (data type)

Syntax **`Double`**

Description Used to declare variables capable of holding real numbers with 15–16 digits of precision. Double variables are used to hold numbers within the following ranges:

Sign	Range
Negative	–1.797693134862315E308 <= **`double`** <= –4.94066E-324
Positive	4.94066E-324 <= **`double`** <= 1.797693134862315E308

The type-declaration character for **`Double`** is #.

Storage

Internally, doubles are 8-byte (64-bit) IEEE values. Thus, when appearing within a structure, doubles require 8 bytes of storage. When used with binary or random files, 8 bytes of storage are required.

Each **`Double`** consists of the following

- A 1-bit sign
- An 11-bit exponent
- A 53-bit significant (mantissa)

See Also Keywords, Data Types, Operators, and Expressions on page 6

DropListBox

Syntax **`DropListBox x, y, width, height, ArrayVariable, .Identifier`**

Description Creates a drop listbox within a dialog template. When the dialog is invoked, the drop listbox will be filled with the elements contained in **`ArrayVariable`**. Drop listboxes are similar to combo boxes, with the following exceptions:

- The listbox portion of a drop listbox is not opened by default. The user must open it by clicking the down arrow.
- The user cannot type into a drop listbox. Only items from the listbox may be selected. With combo boxes, the user can type the name of an item from the list directly or type the name of an item that is not contained within the combo box.

This statement can only appear within a dialog template (i.e., between the **`Begin Dialog`** and **`End Dialog`** statements).

The **`DropListBox`** statement requires the following parameters:

Parameter	Description
x, **y**	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
width, **height**	Integer coordinates specifying the dimensions of the control in dialog units.
ArrayVariable	Single-dimensioned array used to initialize the elements of the drop listbox. If this array has no dimensions, then the drop listbox will be initialized with no elements. A runtime error results if the specified array contains more than one dimension. **ArrayVariable** can specify an array of any fundamental data type (structures are not allowed). null and empty values are treated as zero-length strings.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). This parameter also creates an integer variable whose value corresponds to the index of the drop listbox's selection (0 is the first item, 1 is the second, and so on). This variable can be accessed using the following syntax: **DialogVariable.Identifier**

Example

```
Sub Main
  Dim FieldNames$(4)
  FieldNames$(0) = "Last Name"
  FieldNames$(1) = "First Name"
  FieldNames$(2) = "Zip Code"
  FieldNames$(3) = "State"
  FieldNames$(4) = "City"
  Begin Dialog FindTemplate 16,32,168,48,"Find"
    Text 8,8,37,8,"&Find what:"
    DropListBox 48,6,64,80,FieldNames,.WhichField
    OKButton 120,7,40,14
    CancelButton 120,27,40,14
  End Dialog
  Dim FindDialog As FindTemplate
  FindDialog.WhichField = 1
  Dialog FindDialog
End Sub
```

See Also User Interaction on page 16

E

End

Syntax **`End`**

Description Terminates execution of the current macro, closing all open files.

Example

```
Sub Main
  Session.Echo "The next line will terminate the macro."
  End
End Sub
```

See Also Macro Control and Compilation on page 10

Environ, Environ$

Syntax **`Environ[$](variable$ | VariableNumber)`**

Description Returns the value of the specified environment variable.

`Environ$` returns a **`String`**, whereas **`Environ`** returns a **`String`** variant.

If **`variable$`** is specified, then this function looks for that **`variable$`** in the environment. If the **`variable$`** name cannot be found, then a zero-length string is returned.

If **`VariableNumber`** is specified, then this function looks for the **`N`**th variable within the environment (the first variable being number 1). If there is no such environment variable, then a zero-length string is returned. Otherwise, the entire entry from the environment is returned in the following format:

`variable = value`

Example

```
Sub Main
  Dim a$(1)
  a$(1) = Environ$("COMSPEC")
  Session.Echo "The DOS Comspec variable is set to: " & a$(1)
End Sub
```

See Also Operating System Control on page 15

EOF

Syntax **`EOF(filenumber)`**

Description Returns **`True`** if the end-of-file has been reached for the given file; returns **`False`** otherwise. The **`filenumber`** parameter is an **`Integer`** used to refer to the open file—the number passed to the **`Open`** statement.

With sequential files, **`EOF`** returns **`True`** when the end of the file has been reached (i.e., the next file read command will result in a runtime error).

With Random or Binary files, **`EOF`** returns **`True`** after an attempt has been made to read beyond the end of the file. Thus, **`EOF`** will only return **`True`** when **`Get`** was unable to read the entire record.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Dim s$
  Open "c:\autoexec.bat" For Input As #1
  Do While Not EOF(1)
    Input #1,s$
  Loop
  Close
   Session.Echo "The last line was:" & crlf & s$
End Sub
```

See Also Drive, Folder, and File Access on page 4

Eqv

Syntax **`result = expression1 Eqv expression2`**

Description Performs a logical or binary equivalence on two expressions. If both expressions are either **`Boolean`**, **`Boolean`** variants, or **`Null`** variants, then a logical equivalence is performed as follows:

Expression One	Expression Two	Result
True	True	True
True	False	False
False	True	False
False	False	True

If either expression is **`Null`**, then **`Null`** is returned.

Binary equivalence

If the two expressions are **`Integer`**, then a binary equivalence is performed, returning an **`Integer`** result. All other numeric types (including **`Empty`** variants) are converted to **`Long`** and a binary equivalence is then performed, returning a **`Long`** result.

Binary equivalence forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions, according to the following table:

Bit in Expression One	Bit in Expression Two	Result
1	1	1
0	1	0
1	0	0
0	0	1

Example This example assigns False to **`a`**, performs some equivalent operations, and displays the result. Since **`a`** is equivalent to False, and False is equivalent to 0, and by definition, **`a`** = 0, then the prompt will display "**`A is False.`**"

```
Sub Main
  a = False
  If ((a Eqv False) And (False Eqv 0) And (a = 0)) Then
    Session.Echo "a is False."
  Else
    Session.Echo "a is True."
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Erase

Syntax **`Erase array1 [,array2]...`**

Description Erases the elements of the specified arrays. For dynamic arrays, the elements are erased, and the array is redimensioned to have no dimensions (and therefore no elements). For fixed arrays, only the elements are erased; the array dimensions are not changed.

After a dynamic array is erased, the array will contain no elements and no dimensions. Thus, before the array can be used by your program, the dimensions must be reestablished using the **`Redim`** statement.

Up to 32 parameters can be specified with the **`Erase`** statement.

The meaning of erasing an array element depends on the type of the element being erased:

Element Type	Effect of Erase
Integer	Sets element to 0.
Boolean	Sets element to **`False`**.
Long	Sets element to 0.
Double	Sets element to 0.0.
Date	Sets element to December 30, 1899.
Single	Sets element to 0.0.
String (variable-length)	Frees string, then sets element to a zero-length string.
String (fixed-length)	Sets every character of each element to zero (**`Chr$(0)`**).
Object	Decrements reference count and sets element to **`Nothing`**.
Variant	Sets element to **`empty`**.
User-defined type	Sets each structure element as a separate variable.

Example

```
Sub Main
  Dim a$(10)            'Declare an array.
  a$(1) = Dir$("*")          'Fill element 1 with a filename
 Session.Echo "Array before Erase: " & a$(1)                       'Display element
1.
  Erase a$              'Erase all elements in array
  Session.Echo "Array after Erase: " & a$(1)  'again (should be erased).
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Err (object)

The **`Err`** object allows you to create your own routines to handle errors returned by the compiler, OLE objects, and external DLLs. You can also construct macro code to raise errors as necessary. The methods and properties of the **`Err`** object provide access to the calling OLE object or external DLL, and the source if possible.

Erl

Syntax **`Erl[()]`**

Description Returns the line number of the most recent error. The first line of the macro is 1, the second line is 2, and so on.

The internal value of **`Erl`** is reset to 0 with any of the following statements: **`Resume`**, **`Exit Sub`**, **`Exit Function`**. Thus, if you want to use this value outside an error handler, you must assign it to a variable.

Example

```
Sub Main
  Dim i As Integer
  On Error Goto Trap1
  i = 32767        'Generate an error--overflow.
```

```
  i = i + 1
  Exit Sub
Trap1:
  Session.Echo "Error on line: " & Erl
  Exit Sub        'Reset the error handler.
End Sub
```

See Also Error Handling (topic).

Err.Clear

Syntax **`Err.Clear`**

Description Clears the properties of the **`Err`** object. After this method has been called, the properties of the **`Err`** object will have the following values:

Value	Property
""	**`Err.Description`**
0	**`Err.HelpContext`**
""	**`Err.HelpFile`**
0	**`Err.LastDLLError`**
0	**`Err.Number`**
""	**`Err.Source`**

The properties of the **`Err`** object are automatically reset when any of the following statements are executed: **`Resume`**, **`Exit Function`**, **`On Error`**, **`Exit Sub`**

Example

```
Sub Main
  Dim x As Integer
  On Error Resume Next
  x = InputBox("Type in a number")
  If Err.Number <> 0 Then
    Err.Clear
    x = 0
  End If
  Session.Echo x
End Sub
```

See Also Macro Control and Compilation on page 10

Err.Description

Syntax **`Err.Description [= stringexpression]`**

Description Sets or retrieves the description of the error. For errors generated by the compiler, the **`Err.Description`** property is automatically set. For user-defined errors, you should set this property to be a description of your error. If you set the **`Err.Number`** property to one of the internal error numbers and you don’t set the **`Err.Description`** property, then the **`Err.Description`** property is automatically set when the error is generated (i.e., with **`Err.Raise`**).

Example

```
Sub Main
  Dim x As Integer
  On Error Resume Next
  x = InputBox("Type in a number")
  If Err.Number <> 0 Then
    Session.Echo "The following error occurred: " & Err.Description
    x = 0
  End If
  Session.Echo x
End Sub
```

See Also Macro Control and Compilation on page 10

Err.HelpContext

Syntax **`Err.HelpContext [= contextid]`**

Description Sets or retrieves the help context ID that identifies the help topic for information on the error. The **`Err.HelpContext`** property, together with the **`Err.HelpFile`** property, contain sufficient information to display help for the error. When the compiler generates an error, the **`Err.HelpContext`** property is set to 0 and the and the **`Err.HelpFile`** property is set to ""; the value of the **`Err.Number`** property is sufficient for displaying help in this case. The exception is with errors generated by an OLE automation server; both the **`Err.HelpFile`** and **`Err.HelpContext`** properties are set by the server to values appropriate for the generated error.

When generating your own user-define errors, you should set the **`Err.HelpContext`** property and the **`Err.HelpFile`** property appropriately for your error. If these are not set, then the compiler displays its own help at an appropriate place.

Example

```
Function InputInteger(Prompt,Optional Title,Optional Def)
  On Error Resume Next
  Dim x As Integer
  x = InputBox(Prompt,Title,Def)
  If Err.Number Then
    Err.HelpContext = "WIDGET.HLP"
    Err.HelpContext = 10
    Err.Description = "Integer value expected"
    InputInteger = Null
    Err.Raise 3000
  End If
  InputInteger = x
End Function

Sub Main
  Dim x As Integer
  Do
    On Error Resume Next
    x = InputInteger("Enter a number:")
  Loop Until Err.Number <> 3000
End Sub
```

See Also Macro Control and Compilation on page 10; User Interaction on page 16

Err.HelpFile

Syntax **`Err.HelpFile [= filename]`**

Description Sets or retrieves the name of the help file associated with the error. The **`Err.HelpFile`** property, together with the **`Err.HelpContents`** property, contain sufficient information to display help for the error. When the compiler generates an error, the **`Err.HelpContents`** property is set to 0 and the and the **`Err.HelpFile`** property is set to ""; the value of the **`Err.Number`** property is sufficient for displaying help in this case. The exception is with errors generated by an OLE automation server; both the **`Err.HelpFile`** and **`Err.HelpContext`** properties are set by the server to values appropriate for the generated error.

When generating your own user-defined errors, set the **`Err.HelpContext`** property and the **`Err.HelpFile`** property appropriately for your error. If these are not set, then the compiler displays its own help at an appropriate place.

The **`Err.HelpFile`** property can be set to any valid Windows help file (i.e., a file with a .HLP extension compatible with the WINHELP help engine).

Example

```
Function InputInteger(Prompt,Optional Title,Optional Def)
  On Error Resume Next
  Dim x As Integer
  x = InputBox(Prompt,Title,Def)
  If Err.Number Then
    Err.HelpContext = "WIDGET.HLP"
    Err.HelpContext = 10
    Err.Description = "Integer value expected"
    InputInteger = Null
    Err.Raise 3000
  End If
  InputInteger = x
End Function

Sub Main
  Dim x As Integer
  Do
    On Error Resume Next
    x = InputInteger("Enter a number:")
  Loop Until Err.Number <> 3000
End Sub
```

See Also Macro Control and Compilation on page 10; User Interaction on page 16

Err.LastDLLError

Syntax **`Err.LastDLLError`**

Description Returns the last error generated by an external call—i.e., a call to a routine declared with the **`Declare`** statement that resides in an external module. The **`Err.LastDLLError`** property is automatically set when calling a routine defined in an external module. If no error occurs within the external call, then this property will automatically be set to 0. This property is set by DLL routines that set the last error using the function **`SetLastError()`**. The compiler uses the function **`GetLastError()`** to retrieve the value of this property. The value 0 is returned when calling DLL routines that do not set an error.

Example

```
Declare Sub GetCurrentDirectoryA Lib "kernel32" (ByVal DestLen As Integer, _
ByVal lpDest As String)

Sub Main
  Dim dest As String * 256
  Err.Clear
  GetCurrentDirectoryA len(dest),dest
  If Err.LastDLLError <> 0 Then
    Session.Echo "Error " & Err.LastDLLError & " occurred."
  Else
    Session.Echo "Current directory is " & dest
  End If
End Sub
```

See Also Macro Control and Compilation on page 10

Err.Number

Syntax **`Err.Number [= errornumber]`**

Description Returns or sets the number of the error. The **`Err.Number`** property is set automatically when an error occurs. This property can be used within an error trap to determine which error occurred. You can set the **`Err.Number`** property to any **`Long`** value.

The **`Number`** property is the default property of the **`Err`** object. This allows you to use older style syntax such as those shown below:

```
Err = 6
If Err = 6 Then Session.Echo "Overflow"
```

The **`Err`** function can only be used while within an error trap.

The internal value of the **`Err.Number`** property is reset to 0 with any of the following statements: **`Resume`**, **`Exit Sub`**, **`Exit Function`**. Thus, if you want to use this value outside an error handler, you must assign it to a variable.

Setting **`Err.Number`** to –1 has the side effect of resetting the error state. This allows you to perform error trapping within an error handler. The ability to reset the error handler while within an error trap is not standard Basic. Normally, the error handler is reset only with the **`Resume`**, **`Exit Sub`**, **`Exit Function`**, **`End Function`**, or **`End Sub`** statements.

Example

```
Sub Main
  On Error Goto TestError
  Error 10
  Session.Echo "The returned error is: '" & Err() & " - " & _
    Error$ & "'"
  Exit Sub
TestError:
  If Err = 55 Then       'File already open.
```

```
    Session.Echo "Cannot copy an open file. Close it and try again."
  Else
    Session.Echo "Error '" & Err & "' has occurred!"
    Err = 999
  End If
  Resume Next
End Sub
```

See Also Macro Control and Compilation on page 10

Err

Syntax **`Err = value`**

Description Sets the value returned by the **`Err`** function to a specific **`Integer`** value. Only positive values less than or equal to 32767 can be used. Setting **`value`** to –1 has the side effect of resetting the error state. This allows you to perform error trapping within an error handler. The ability to reset the error handler while within an error trap is not standard Basic. Normally, the error handler is reset only with the **`Resume`**, **`Exit Sub`**, or **`Exit Function`** statement.

Example

```
Sub Main
  On Error Goto TestError
  Error 10
  Session.Echo "The returned error is: '" & Err() & " - " & Error$ & "'"
  Exit Sub
TestError:
  If Err = 55 Then              'File already open.
    Session.Echo "Cannot copy an open file. Close it and try again."
  Else
    Session.Echo "Error '" & Err & "' has occurred."
    Err = 999
  End If
  Resume Next
End Sub
```

See Also Macro Control and Compilation on page 10

Err.Raise

Syntax **`Err.Raise number [,[source] [,[description] [,[helpfile] [,helpcontext]]]]`**

Description Generates a runtime error, setting the specified properties of the **`Err`** object. The **`Err.Raise`** method has the following named parameters:

Parameter	Description
number	A **Long** value indicating the error number to be generated. This parameter is required. Predefined errors are in the range 0 to 1000.
Source	An optional **String** expression specifying the source of the error—i.e., the object or module that generated the error. If omitted, then the compiler uses the name of the currently executing macro.
description	An optional **String** expression describing the error. If omitted and **number** maps to a predefined error number, then the corresponding predefined description is used. Otherwise, the error "Application-defined or object-define error" is used.
helpfile	An optional **String** expression specifying the name of the help file containing context-sensitive help for this error. If omitted and number maps to a predefined error number, then the default help file is assumed.
Helpcontext	An optional long value specifying the topic within **helpfile** containing context-sensitive help for this error. If some arguments are omitted, then the current property values of the **Err** object are used.

This method can be used in place of the **Error** statement for generating errors. Using the **Err.Raise** method gives you the opportunity to set the desired properties of the **Err** object in one statement.

Example

```
Sub Main
  Dim x As Variant
  On Error Goto TRAP
  x = InputBox("Enter a number:")
  If Not IsNumeric(x) Then
    Err.Raise 3000,,"Invalid number specified","WIDGET.HLP",30
  End If
  Session.Echo x
  Exit Sub
TRAP:
  Session.Echo Err.Description
End Sub
```

See Also Macro Control and Compilation on page 10

Err.Source

Syntax **Err.Source [= stringexpression]**

Description Sets or retrieves the source of a runtime error.

For OLE automation errors generated by the OLE server, the **Err.Source** property is set to the name of the object that generated the error. For all other errors generated by the macro language, the **Err.Source** property is automatically set to be the name of the macro that generated the error.

For user-defined errors, the **`Err.Source`** property can be set to any valid string expression indicating the source of the error. If the **`Err.Source`** property is not explicitly set for user-defined errors, the value is the name of the macro in which the error was generated.

Example

```
Function InputInteger(Prompt,Optional Title,Optional Def)
  On Error Resume Next
  Dim x As Integer
  x = InputBox(Prompt,Title,Def)
  If Err.Number Then
    Err.Source = "InputInteger"
    Err.Description = "Integer value expected"
    Err.Raise 3000
  End If
  InputInteger = x
End Function

Sub Main
  On Error Resume Next
  x = InputInteger("Enter a number:")
  If Err.Number Then Session.Echo Err.Source & ":" & Err.Description
End Sub
```

See Also Macro Control and Compilation on page 10

Error Handling (topic)

The macro language supports nested error handlers. When an error occurs within a subroutine, the compiler checks for an **`On Error`** handler within the currently executing subroutine or function. An error handler is defined as follows:

```
Sub foo()
  On Error Goto catch
  'Do something here.
  Exit Sub
catch:
  'Handle error here.
End Sub
```

Error handlers have a life local to the procedure in which they are defined. The error is reset when any of the following conditions occurs:

- An **`On Error`** or **`Resume`** statement is encountered.
- When **`Err.Number`** is set to -1.
- When the **`Err.Clear`** method is called.
- When an **`Exit Sub`**, **`Exit Function`**, **`End Function`**, **`End Sub`** is encountered.

Cascading Errors

If a runtime error occurs and no **`On Error`** handler is defined within the currently executing procedure, then control returns to the calling procedure and the error handler there runs. This process repeats until a procedure is found that contains an error handler or until there are no more procedures. If an error is not trapped or if an error occurs within the error handler, then there is an error message, halting execution of the macro.

Once an error handler has control, it should address the condition that caused the error and resume execution with the **`Resume`** statement. This statement resets the error handler, transferring execution to an appropriate place within the current procedure. The error is reset if the procedure exits without first executing **`Resume`**.

Visual Basic Compatibility

Where possible, the macro language has the same error numbers and error messages as Visual Basic. This is useful for porting macros between environments.

Handling errors involves querying the error number or error text using the **`Error$`** function or **`Err.Description`** property. Since this is the only way to handle errors, compatibility with Visual Basic's error numbers and messages is essential.

Macro language errors fall into three categories:

- **Visual Basic-compatible errors:** These errors, numbered between 0 and 799, are numbered and named according to the errors supported by Visual Basic.
- **Macro language errors:** These errors, numbered from 800 to 999, are unique to the macro language.
- **User-defined errors:** These errors, equal to or greater than 1,000, are available for use by extensions or by the macro itself.

You can intercept trappable errors using the **`On Error`** construct. Almost all errors are trappable except for various system errors.

Error, Error$ (functions)

Syntax **`Error[$][(errornumber)]`**

Description Returns a **`String`** containing the text corresponding to the given error number or the most recent error.

`Error$` returns a **`String`**, whereas **`Error`** returns a **`String`** variant.

The **`errornumber`** parameter is an **`Integer`** containing the number of the error message to retrieve. If this parameter is omitted, then the function returns the text corresponding to the most recent runtime error (i.e., the same as returned by the **`Err.Description`** property). If no runtime error has occurred, then a zero-length string is returned.

If the **`Error`** statement was used to generate a user-defined runtime error, then this function will return a zero-length string ("").

Example

```
Sub Main
  On Error Goto TestError
  Error 10
  Session.Echo "The returned error is: '" & Err() & " - " & Error$ & "'"
  Exit Sub
TestError:
  If Err = 55 Then       'File already open.
    Session.Echo "Cannot copy an open file. Close it and try again."
  Else
    Session.Echo "Error '" & Err & "' has occurred."
    Err = 999
  End If
  Resume Next
End Sub
```

See Also Character and String Manipulation on page 3; Macro Control and Compilation on page 10

Error (statement)

Syntax **`Error errornumber`**

Description Simulates the occurrence of the given runtime error. The **`errornumber`** parameter is any **`Integer`** containing either a built-in error number or a user-defined error number. The **`Err.Number`** property can be used within the error trap handler to determine the value of the error.

The **`Error`** statement is provided for backward compatibility. Use the **`Err.Raise`** method instead. When using the **`Error`** statement to generate an error, the **`Err`** object's properties are set to the following default values:

Property	Default Value
`Number`	**`errornumber`** as specified in the **`Error`** statement.
`Source`	Name of currently executing macro.
`Description`	Text of error. If **`errornumber`** is unknown, is set to an empty string.
`HelpFile`	Name of help file.
`HelpContext`	Context ID corresponding to **`errornumber`**.

Example

```
Sub Main
  On Error Goto TestError
  Error 10
  Session.Echo "The returned error is: '" & Err & " - " & Error$ & "'"
  Exit Sub
TestError:
  If Err = 55 Then       'File already open.
    Session.Echo "Cannot copy an open file. Close it and try again."
  Else
    Session.Echo "Error '" & Err & "' has occurred."
    Err = 999
  End If
  Resume Next
End Sub
```

See Also Macro Control and Compilation on page 10

Exit Do

Syntax `Exit Do`

Description Causes execution to continue on the statement following the **`Loop`** clause. This statement can only appear within a **`Do...Loop`** statement.

Example

```
Const crlf = Chr$(13) + Chr$(10)
Sub Main
  Dim a$(5)
  Do
     i% = i% + 1
    If i% = 1 Then
      a(i%) = Dir$("*")
    Else
       a(i%) = Dir$
    End If
    If i% >= 10 Then Exit Do
  Loop While (a(i%) <> "")
  If i% = 10 Then
    Session.Echo i% & " entries processed!"
  Else
    Session.Echo "Less than " & i% & " entries processed!"
  End If
End Sub
```

See Also Macro Control and Compilation on page 10

Exit For

Syntax `Exit For`

Description Causes execution to exit the innermost **`For`** loop, continuing execution on the line following the **`Next`** statement. This statement can only appear within a **`For...Next`** block.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Dim a$(100)
  For i = 1 To 100
    If i = 1 Then
      a$(i) = Dir$("*")
    Else
      a$(i) = Dir$
    End If
    If (a$(i) = "") Or (i >= 100) Then Exit For
  Next i
  mesg = "There are " & i & " files found." & crlf
  Session.Echo mesg & a$(1) & crlf & a$(2) & crlf & a$(3) & crlf & a$(10)
End Sub
```

See Also Macro Control and Compilation on page 10

Exit Function

Syntax `Exit Function`

Description Causes execution to exit the current function, continuing execution on the statement following the call to this function. This statement can only appear within a function.

Example

```
Function Test_Exit() As Integer
  Session.Echo "Testing function exit, returning to Main()."
  Test_Exit = 0
  Exit Function
  Session.Echo "This line should never execute."
End Function

Sub Main
  a% = Test_Exit()
  Session.Echo "This is the last line of Main()."
End Sub
```

See Also Macro Control and Compilation on page 10

Exit Sub

Syntax `Exit Sub`

Description Causes execution to exit the current subroutine, continuing execution on the statement following the call to this subroutine. This statement can appear anywhere within a subroutine. It cannot appear within a function.

Example

```
Sub Main
  Session.Echo "Terminating Main()."
  Exit Sub
  Session.Echo "Still here in Main()."
End Sub
```

See Also Macro Control and Compilation on page 10

Exp

Syntax `Exp(number)`

Description Returns the value of **`e`** raised to the power of **`number`**. The **`number`** parameter is a **`Double`** within the following range:

`0 <= number <= 709.782712893.`

A runtime error is generated if **`number`** is out of the range specified above.

The value of **`e`** is 2.71828.

Example

```
Sub Main
  a# = Exp(12.40)
  Session.Echo "e to the 12.4 power is: " & a#
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

Expression Evaluation (topic)

Expressions may involve data of different types. When this occurs, the two arguments are converted to be of the same type by promoting the less precise operand to the same type as the more precise operand. For example, the compiler will promote the value of i% to a double in the following expression:

```
result# = i% * d#
```

In some cases, the data type to which each operand is promoted is different than that of the most precise operand. This is dependent on the operator and the data types of the two operands and is noted in the description of each operator.

If an operation is performed between a numeric expression and a **`String`** expression, then the **`String`** expression is usually converted to be of the same type as the numeric expression. For example, the following expression converts the **`String`** expression to an **`Integer`** before performing the multiplication:

```
result = 10 * "2"        'Result is equal to 20.
```

There are exceptions to this rule, as noted in the description of the individual operators.

Type Coercion

The compiler performs numeric type conversion automatically. Automatic conversions sometimes result in overflow errors, as shown in the following example:

```
d# = 45354
i% = d#
```

In this example, an overflow error is generated because the value contained in **`d#`** is larger than the maximum size of an **`Integer`**.

Rounding

When floating-point values (**`Single`** or **`Double`**) are converted to integer values (**`Integer`** or **`Long`**), the fractional part of the floating-point number is lost, rounding to the nearest integer value. The macro language uses Baker's rounding:

- If the fractional part is larger than .5, the number is rounded up.
- If the fractional part is smaller than .5, the number is rounded down.

- If the fractional part is equal to .5, then the number is rounded up if it is odd and down if it is even.

The following table shows sample values before and after rounding:

Before Rounding	After Rounding
2.1	2
4.6	5
2.5	2
3.5	4

Default Properties

When an OLE object variable or an **`Object`** variant is used with numerical operators such as addition or subtraction, then the default property of that object is automatically retrieved. For example, consider the following:

```
Dim Excel As Object
Set Excel = GetObject(,"Excel.Application")
Session.Echo "This application is " & Excel
```

The above example displays "This application is Microsoft Excel". When the variable Excel is used within the expression, the default property is automatically retrieved, which, in this case, is the string "Microsoft Excel." Considering that the default property of the Excel object is .Value, then the following two statements are equivalent:

```
Session.Echo "This application is " & Excel
Session.Echo "This application is " & Excel.Value
```

F

FileAttr

Syntax `FileAttr(filenumber, returntype)`

Description Returns an **`Integer`** specifying the file mode (if **`returntype`** is 1) or the operating system file handle (if **`returntype`** is 2). The **`FileAttr`** function takes the following named parameters:

Parameter	Description
`filenumber`	Integer value used to refer to the open file—the number passed to the Open statement.
`Returntype`	Integer specifying the type of value to be returned. If returntype is 1, then one of the following values is returned: **`1`** Input **`2`** Output **`4`** Random **`6`** Append **`32`** Binary

If **`returntype`** is 2, then the operating system file handle is returned. This is a special **`Integer`** value identifying the file.

Example

```
Sub Main
  Open "c:\autoexec.bat" For Input As #1
  a% = FileAttr(1,1)
  Select Case a%
    Case 1
      Session.Echo "Opened for input."
    Case 2
      Session.Echo "Opened for output."
    Case 4
      Session.Echo "Opened for random."
    Case 8
      Session.Echo "Opened for append."
```

```
    Case 32
      Session.Echo "Opened for binary."
    Case Else
      Session.Echo "Unknown file mode."
    End Select
  a% = FileAttr(1,2)
  Session.Echo "File handle is: " & a%
  Close
End Sub
```

See Also Drive, Folder, and File Access on page 4

FileCopy

Syntax `FileCopy source, destination`

Description Copies a **`source`** file to a **`destination`** file. The **`FileCopy`** function takes the following named parameters:

Parameter	Description
`source`	String containing the name of a single file to copy. The source parameter cannot contain wildcards (? or *) but may contain path information.
`Destination`	String containing a single, unique destination file, which may contain a drive and path specification.

The file will be copied and renamed if the **`source`** and **`destination`** filenames are not the same.

Example

```
Sub Main
  On Error Goto ErrHandler
  FileCopy "c:\autoexec.bat", "c:\autoexec.sav"
  Open "c:\autoexec.sav" For Input As # 1
  FileCopy "c:\autoexec.sav", "c:\autoexec.sv2"
  Close
  Exit Sub
ErrHandler:
  If Err = 55 Then       'File already open.
    Session.Echo "Cannot copy an open file. Close it and try again."
  Else
    Session.Echo "An unspecified file copy error has occurred."
  End If
  Resume Next
End Sub
```

See Also Drive, Folder, and File Access on page 4

FileDateTime

Syntax `FileDateTime(pathname)`

Description Returns a **`Date`** variant representing the date and time of the last modification of a file. This function retrieves the date and time of the last modification of the file specified by **`pathname`** (wildcards are not allowed). A runtime error results if the file does not exist. The value returned can be used with the date/time functions (i.e., **Year**, **Month**, **Day**, **Weekday**, **Minute**, **Second**, **Hour**) to extract the individual elements.

Win32 stores the file creation date, last modification date, and the date the file was last written to. The **FileDateTime** function only returns the last modification date.

Example

```
Sub Main
  If FileExists("c:\autoexec.bat") Then
    a# = FileDateTime("c:\autoexec.bat")
    Session.Echo "The date/time information for the file is: " & Year(a#) & "-" &
Month(a#) & "-" & Day(a#)
  Else
    Session.Echo "The file does not exist."
  End If
End Sub
```

See Also Drive, Folder, and File Access on page 4; Time and Date Access on page 17

FileDirs

Syntax `FileDirs array() [,dirspec$]`

Description Fills a **String** or **Variant** array with directory names from disk. The **FileDirs** statement takes the following parameters:

Parameter	Description
array()	Either a zero- or a one-dimensioned array of strings or variants. The array can be either dynamic or fixed. If **array()** is dynamic, then it will be redimensioned to exactly hold the new number of elements. If there are no elements, then the array will be redimensioned to contain no dimensions. You can use the **LBound**, **UBound**, and **ArrayDims** functions to determine the number and size of the new array's dimensions.
array()	If the array is fixed, each array element is first erased, then the new elements are placed into the array. If there are fewer elements than will fit in the array, then the remaining elements are initialized to zero-length strings (for string arrays) or **Empty** (for variant arrays). A runtime error results if the array is too small to hold the new elements.
dirspec$	String containing the file search mask, such as: **t*.c:*.*** If this parameter is omitted or an empty string, then * is used, which fills the array with all the subdirectory names within the current directory.

Example
```
Sub Main
  Dim a$()
  FileDirs a$,"c:\*.*"
  Session.Echo "The first directory is: " & a$(0)
End Sub
```

See Also Character and String Manipulation on page 3; Drive, Folder, and File Access on page 4

FileExists

Syntax `FileExists(filename$)`

Description Returns **`True`** if **`filename$`** exists; returns **`False`** otherwise. This function determines whether a given **`filename$`** is valid. This function returns **`False`** if **`filename$`** specifies a subdirectory.

Example
```
Sub Main
  If FileExists("c:\autoexec.bat") Then
    Session.Echo "This file exists!"
  Else
    Session.Echo "File does not exist."
  End If
End Sub
```

See Also Drive, Folder, and File Access on page 4

FileLen

Syntax `FileLen(pathname)`

Description Returns a **`Long`** representing the length of **`pathname`** in bytes. This function is used in place of the **`LOF`** function to retrieve the length of a file without first opening the file. A runtime error results if the file does not exist.

Example
```
Sub Main
  If (FileExists("c:\autoexec.bat") And (FileLen("c:\autoexec.bat") _
<> 0)) Then
    b% = FileLen("c:\autoexec.bat")
    Session.Echo "The length of autoexec.bat is: " & b%
  Else
    Session.Echo "File does not exist."
  End If
End Sub
```

See Also Drive, Folder, and File Access on page 4

FileList

Syntax `FileList array() [,[filespec$] [,[include_attr] [,exclude_attr]]]`

Description Fills a **`String`** or **`Variant`** array with filenames from disk. The **`FileList`** function takes the following parameters:

Parameter	Description
`array()`	Either a zero- or a one-dimensioned array of strings or variants. The array can be either dynamic or fixed. If **`array()`** is dynamic, then it will be redimensioned to exactly hold the new number of elements. If there are no elements, then the array will be redimensioned to contain no dimensions. You can use the **`LBound`**, **`UBound`**, and **`ArrayDims`** functions to determine the number and size of the new array's dimensions. If the array is fixed, each array element is first erased, then the new elements are placed into the array. If there are fewer elements than will fit in the array, then the remaining elements are initialized to zero-length strings (for string arrays) or **`Empty`** (for variant arrays). A runtime error results if the array is too small to hold the new elements.
`Filespec$`	String specifying which filenames are to be included in the list. The **`filespec$`** parameter can include wildcards, such as * and ?. If this parameter is omitted, then * is used.
`include_attr`	Integer specifying attributes of files you want included in the list. It can be any combination of the attributes listed below.
`exclude_attr`	Integer specifying attributes of files you want excluded from the list. It can be any combination of the attributes listed below.

The **`FileList`** function returns different files as specified by the **`include_attr`** and **`exclude_attr`** and whether these parameter have been specified. The following table shows these differences: If neither the **`include_attr`** or **`exclude_attr`** has been specified, then the following defaults are assumed:

Parameter	Default
`exclude_attr`	**`ebHidden`** Or **`ebDirectory`** Or **`ebSystem`** Or **`ebVolume`**
`include_attr`	**`ebNone`** Or **`ebArchive`** Or **`ebReadOnly`**

If **`include_attr`** is specified and **`exclude_attr`** is missing, then **`FileList`** excludes all files not specified by **`include_attr`**. If **`include_attr`** is missing, its value is assumed to be zero.

Wildcards

The **`*`** character matches any sequence of zero or more characters, whereas the **`?`** character matches any single character. Multiple **`*`**'s and **`?`**'s can appear within the expression to form complete searching patterns. The following table shows some examples:

This Pattern	Matches These Files	Not These Files
*S.*TXT	SAMPLE. TXT, GOOSE.TXT, SAMS.TXT	SAMPLE, SAMPLE.DAT
C*T.TXT	CAT.TXT	CAP.TXT, ACATS.TXT
C*T	CAT, CAP.TXT	CAT.DOC
C?T	CAT, CUT	CAT.TXT, CAPITCT
*	(All files)	

File attributes

These numbers can be any combination of the following:

Constant	Value	Includes
ebNormal	0	Read-only, archive, subdir, none
ebReadOnly	1	Read-only files
ebHidden	2	Hidden files
ebSystem	4	System files
ebVolume	8	Volume label
ebDirectory	16	Subdirectories
ebArchive	32	Files that have changed since the last backup
ebNone	64	Files with no attributes

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Dim a$()
  FileList a$,"*.*", (ebNormal + ebNone), ebSystem
  If ArrayDims(a$) > 0 Then
     Session.Echo a$(1) & crlf & a$(2) & crlf & a$(3) & crlf & a$(4)
  Else
    Session.Echo "No files found."
  End If
End Sub
```

See Also Drive, Folder, and File Access on page 4

FileParse$

Syntax **FileParse$(filename$[, operation])**

Description Returns a **String** containing a portion of **filename$** such as the path, drive, or file extension. The **filename$** parameter can specify any valid filename (it does not have to exist). For example:

```
..\test.dat
c:\sheets\test.dat
test.dat
```

A runtime error is generated if **filename$** is a zero-length string.

The optional **operation** parameter is an **Integer** specifying which portion of the **filename$** to extract. It can be any of the following values.

Value	Meaning	Example
0	Full name	c:\sheets\test.dat
1	Drive	c
2	Path	c:\sheets
3	Name	test.dat
4	Root	test
5	Extension	dat

If **operation** is not specified, then the full name is returned. A runtime error will result if **operation** is not one of the above values.

A runtime error results if **filename$** is empty.

Note The backslash and forward slash can be used interchangeably. For example, "c:\test.dat" is the same as "c:/test.dat".

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Dim a$(6)
  For i = 1 To 5
    a$(i) = FileParse$("c:\testsub\autoexec.bat",i - 1)
  Next i
  Session.Echo a$(1) & crlf & a$(2) & crlf & a$(3) & crlf & a$(4) & crlf & a$(5)
End Sub
```

See Also Character and String Manipulation on page 3; Drive, Folder, and File Access on page 4

Fix

Syntax `Fix(number)`

Description Returns the integer part of **number**. This function returns the integer part of the given value by removing the fractional part. The sign is preserved. The **Fix** function returns the same type as **number**, with the following exceptions:

- If **number** is **Empty**, then an **Integer** variant of value 0 is returned.
- If **number** is a **String**, then a **Double** variant is returned.
- If **number** contains no valid data, then a **Null** variant is returned.

Example

```
Sub Main
  a# = -19923.45
  b% = Fix(a#)
  Session.Echo "The fixed portion of -19923.45 is: " & b%
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

For...Each

Syntax

```
For Each member in group:
  [statements]
  [Exit For]
  [statements]
Next [member]
```

Description Repeats a block of statements for each element in a collection or array. The **`For...Each`** statement takes the following parameters:

Parameter	Description
`member`	Name of a variable to hold an element for each iteration of the loop. If group is an array, then member must be a variant variable. If group is a collection, then member must be an object variable, an explicit OLE automation object, or a variant.
`Group`	Name of a collection or array.
`Statements`	Any number of statements.

The compiler supports iteration through OLE collections or arrays with the exception of arrays of user-defined types or fixed-length strings. The iteration variable is a copy of the collection or array element in the sense that change the value of **`member`** within the loop has no effect on the collection or array.

The **`For...Each`** statement traverses array elements in the same order the elements are stored in memory. For example, the array elements contained in the array defined by the statement

```
Dim a(1 To 2,3 To 4)
```

are traversed in the following order: (1,3), (1,4), (2,3), (2,4). The order in which the elements are traversed should not be relevant to the correct operation of the macro.

The **`For...Each`** statement continues executing until there are no more elements in **`group`** or until an **`Exit For`** statement is encountered.

`For...Each` statements can be nested. In such a case, the **`Next`** [**`member`**] statement applies to the innermost **`For...Each`** or **`For...Next`** statement. Each **`member`** variable of nested **`For...Each`** statements must be unique.

A **`Next`** statement appearing by itself (with no **`member`** variable) matches the innermost **`For...Each`** or **`For...Next`** loop.

Due to errors in program logic, you can inadvertently create infinite loops in your code. When you're running a macro within the macro editor, you can break out of an infinite loop by pressing Ctrl+Break.

Example

```
Sub Main
  Dim a(3 To 10) As Single
  Dim i As Variant
  Dim s As String
  For i = 3 To 10
    a(i) = Rnd()
  Next i
  For Each i In a
    i = i + 1
  Next i
  s = ""
  For Each i In a
    If s <> "" Then s = s & ","
    s = s & i
  Next i
  Session.Echo s
End Sub
```

The following subroutine displays the names of each worksheet in an Excel workbook.

```
Sub Main
  Dim Excel As Object
  Dim Sheets As Object
  Set Excel = CreateObject("Excel.Application")
  Excel.Visible = 1
  Excel.Workbooks.Add
  Set Sheets = Excel.Worksheets
  For Each a In Sheets
    Session.Echo a.Name
  Next a
End Sub
```

See Also Macro Control and Compilation on page 10

For...Next

Syntax

```
For counter = start To end [Step increment]
  [statements]
  [Exit For]
  [statements]
Next [counter [,nextcounter]... ]
```

Description Repeats a block of statements a specified number of times, incrementing a loop counter by a given increment each time through the loop. The **`For`** statement takes the following parameters:

Parameter	Description
counter	Name of a numeric variable. Variables of the following types can be used: integer, long, single, double, variant.
Start	Initial value for **counter**. The first time through the loop, **counter** is assigned this value.
End	Final value for **counter**. The **statements** will continue executing until **counter** is equal to **end**.
Increment	Amount added to counter each time through the loop. If **end** is greater than **start**, then **increment** must be positive. If **end** is less than **start**, then **increment** must be negative. If **increment** is not specified, then 1 is assumed. The expression given as **increment** is evaluated only once. Changing the step during execution of the loop will have no effect.
statements	Any number of statements.

The **For...Next** statement continues executing until an **Exit For** statement is encountered when **counter** is greater than **end**.

For...Next statements can be nested. In such a case, the **Next** [**counter**] statement applies to the innermost **For...Next**.

The **Next** clause can be optimized for nested next loops by separating each counter with a comma. The ordering of the counters must be consistent with the nesting order (innermost counter appearing before outermost counter). The following example shows two equivalent **For** statements:

```
For i = 1 To 10        For i = 1 To 10
  For j = 1 To 10      For j = 1 To 10
  Next j           Next j,i
Next i
```

A **Next** clause appearing by itself (with no **counter** variable) matches the innermost **For** loop.

The **counter** variable can be changed within the loop but will have no effect on the number of times the loop will execute.

Due to errors in program logic, you can inadvertently create infinite loops in your code. When you're running a macro within the macro editor, you can break out of an infinite loop by pressing Ctrl+Break.

Example

```
Sub Main
  For x = -1 To 0
    For y = -1 To 0
      Z = x Or y
      mesg = mesg & Format(Abs(x%),"0") & " Or "
      mesg = mesg & Format(Abs(y%),"0") & " = "
```

```
      mesg = mesg & Format(Z,"True/False") & Basic.Eoln$
    Next y
  Next x
  Session.Echo mesg
End Sub
```

See Also Macro Control and Compilation on page 10

Format, Format$

Syntax **`Format[$](expression [, [format] [, [firstdayofweek] [, firstweekofyear]]])`**

Description Returns a **`String`** formatted to user specification. **`Format$`** returns a **`String`**, whereas **`Format`** returns a **`String`** variant. The **`Format$`**/**`Format`** functions take the following named parameters:

Parameter	Description
`expression`	String or numeric expression to be formatted. The compiler will only examine the first 255 characters of **`expression`**.
`format`	Format expression that can be either one of the built-in formats or a user-defined format consisting of characters that specify how the expression should be displayed. string, numeric, and date/time formats cannot be mixed in a single **`format`** expression.
`Firstdayofweek`	Indicates the first day of the week. If omitted, then Sunday is assumed (i.e., the constant ebSunday described below).
`Firstweekofyear`	Indicates the first week of the year. If omitted, then the first week of the year is considered to be that containing January 1 (i.e., the constant ebFirstJan1 as described bellow).

If **`format`** is omitted and the expression is numeric, then these functions perform the same function as the **`Str$`** or **`Str`** statements, except that they do not preserve a leading space for positive values.

If **`expression`** is **`Null`**, then a zero-length string is returned.

The maximum length of the string returned by **`Format`** or **`Format$`** functions is 255.

The **`firstdayofweek`** parameter, if specified, can be any of the following constants:

Constant	Value	Description
`ebUseSystem`	0	Use the system setting for **`firstdayofweek`**.
`EbSunday`	1	Sunday (the default)
`ebMonday`	2	Monday
`ebTuesday`	3	Tuesday
`ebWednesday`	4	Wednesday

Constant	Value	Description
`ebThursday`	5	Thursday
`ebFriday`	6	Friday
`ebSaturday`	7	Saturday

The **`firstdayofyear`** parameter, if specified, can be any of the following constants:

Constant	Value	Description
`ebUseSystem`	0	Use the system setting for **`firstdayofyear`**.
`EbFirstJan1`	1	The first week of the year is that in which January 1 occurs (the default).
`ebFirstFourDays`	2	The first week of the year is that containing at least four days in the year.
`ebFirstFullWeek`	3	The first week of the year is the first full week of the year.

Built-in formats

To format numeric expressions, you can specify one of the built-in formats. There are two categories of built-in formats: one deals with numeric expressions and the other with date/time values. The following tables list the built-in numeric and date/time format strings, followed by an explanation of what each does.

Numeric formats

Format	Description
`General Number`	Displays the numeric expression as is, with no additional formatting.
`Currency`	Displays the numeric expression as currency, with thousands separator if necessary. The built-in currency format allows the specification of an optional user-defined format specification used only for zero values: `Currency;`**`zero-format-string`** where **`zero-format-string`** is a user-defined format used specifically for zero values.
`Fixed`	Displays at least one digit to the left of the decimal separator and two digits to the right.
`Standard`	Displays the numeric expression with thousands separator if necessary. Displays at least one digit to the left of the decimal separator and two digits to the right.
`Percent`	Displays the numeric expression multiplied by 100. A percent sign (%) will appear at the right of the formatted output. Two digits are displayed to the right of the decimal separator.

Format	Description
`Scientific`	Displays the number using scientific notation. One digit appears before the decimal separator and two after.
`Yes/No`	Displays No if the numeric expression is 0. Displays Yes for all other values.
`True/False`	Displays False if the numeric expression is 0. Displays True for all other values.
`On/Off`	Displays Off if the numeric expression is 0. Displays On for all other values.

Date/Time formats

Format	Description
`General date`	Displays the date and time. If there is no fractional part in the numeric expression, then only the date is displayed. If there is no integral part in the numeric expression, then only the time is displayed. Output is in the following form: **`1/1/95 01:00:00 AM`**
`Long date`	Displays a long date—prints out the day of the week, the full name of the month, and the numeric date and year.
`Medium date`	Displays a medium date—prints out only the abbreviated name of the month.
`Short date`	Displays a short date.
`Long time`	Displays the long time. The default is: **`h:mm:ss`**.
`Medium time`	Displays the time using a 12-hour clock. Hours and minutes are displayed, and the AM/PM designator is at the end.
`Short time`	Displays the time using a 24-hour clock. Hours and minutes are displayed.

Default date/time formats are read from the **`[Intl]`** section of the win.ini file.

User-defined formats

In addition to the built-in formats, you can specify a user-defined format by using characters that have special meaning when used in a format expression. The following list the characters you can use for numeric, string, and date/time formats and explain their functions.

Numeric formats

Character	Meaning
Empty string	Displays the numeric expression as is, with no additional formatting.
`0`	This is a digit placeholder. Displays a number or a 0. If a number exists in the numeric expression in the position where the 0 appears, the number will be displayed. Otherwise, a 0 will be displayed. If there are more 0s in the format string than there are digits, the leading and trailing 0s are displayed without modification.
`#`	This is a digit placeholder. Displays a number or nothing. If a number exists in the numeric expression in the position where the number sign appears, the number will be displayed. Otherwise, nothing will be displayed. Leading and trailing 0s are not displayed.
`.`	This is the decimal placeholder. Designates the number of digits to the left of the decimal and the number of digits to the right. The character used in the formatted string depends on the decimal placeholder, as specified by your locale.
`%`	This is the percentage operator. The numeric expression is multiplied by 100, and the percent character is inserted in the same position as it appears in the user-defined format string.
`,`	This is the thousands separator. The common use for the thousands separator is to separate thousands from hundreds. To specify this use, the thousands separator must be surrounded by digit placeholders. Commas appearing before any digit placeholders are specified are just displayed. Adjacent commas with no digit placeholders specified between them and the decimal mean that the number should be divided by 1,000 for each adjacent comma in the format string. A comma immediately to the left of the decimal has the same function. The actual thousands separator character used depends on the character specified by your locale.
`E- E+ e- e+`	These are the scientific notation operators, which display the number in scientific notation. At least one digit placeholder must exist to the left of **`E-`**, **`E+`**, **`e-`**, or **`e+.`** Any digit placeholders displayed to the left of **`E-`**, **`E+`**, **`e-`**, or **`e+`** determine the number of digits displayed in the exponent. Using **`E+`** or **`e+`** places a **+** in front of positive exponents and a **–** in front of negative exponents. Using **`E-`** or **`e-`** places a **–** in front of negative exponents and nothing in front of positive exponents.
`:`	This is the time separator. Separates hours, minutes, and seconds when time values are being formatted. The actual character used depends on the character specified by your locale.
`/`	This is the date separator. Separates months, days, and years when date values are being formatted. The actual character used depends on the character specified by your locale.

Character	Meaning
`- + $ ( ) space`	These are the literal characters you can display. To display any other character, you should precede it with a backslash or enclose it in quotes.
`\`	This designates the next character as a displayed character. To display characters, precede them with a backslash. To display a backslash, use two backslashes. Double quotation marks can also be used to display characters. Numeric formatting characters, date/time formatting characters, and string formatting characters cannot be displayed without a preceding backslash.
`"ABC"`	Displays the text between the quotation marks, but not the quotation marks. To designate a double quotation mark within a format string, use two adjacent double quotation marks.
`*`	This will display the next character as the fill character. Any empty space in a field will be filled with the specified fill character.

Numeric formats can contain one to three parts. Each part is separated by a semicolon. If you specify one format, it applies to all values. If you specify two formats, the first applies to positive values and the second to negative values. If you specify three formats, the first applies to positive values, the second to negative values, and the third to 0s. If you include semicolons with no format between them, the format for positive values is used.

String formats

Character	Meaning
`@`	This is a character placeholder. It displays a character if one exists in the expression in the same position; otherwise, it displays a space. Placeholders are filled from right to left unless the format string specifies left to right.
`&`	This is a character placeholder. It displays a character if one exists in the expression in the same position; otherwise, it displays nothing. Placeholders are filled from right to left unless the format string specifies left to right.
`<`	This character forces lowercase. It displays all characters in the expression in lowercase.
`>`	This character forces uppercase. It displays all characters in the expression in uppercase.
`!`	This character forces placeholders to be filled from left to right. The default is right to left.

Date/Time formats

Character	Meaning
c	Displays the date as **ddddd** and the time as ttttt. Only the date is displayed if no fractional part exists in the numeric expression. Only the time is displayed if no integral portion exists in the numeric expression.
d	Displays the day without a leading 0 (1–31).
dd	Displays the day with a leading 0 (01–31).
ddd	Displays the day of the week abbreviated (Sun–Sat).
dddd	Displays the day of the week (Sunday–Saturday).
ddddd	Displays the date as a short date.
dddddd	Displays the date as a long date.
w	Displays the number of the day of the week (1–7). Sunday is 1; Saturday is 7.
ww	Displays the week of the year (1–53).
m	Displays the month without a leading 0 (1–12). If m immediately follows h or **hh**, **m** is treated as minutes (0–59).
mm	Displays the month with a leading 0 (01–12). If mm immediately follows **h** or **hh**, mm is treated as minutes with a leading 0 (00–59).
mmm	Displays the month abbreviated (Jan–Dec).
mmmm	Displays the month (January–December).
q	Displays the quarter of the year (1–4).
yy	Displays the year, not the century (00–99).
yyyy	Displays the year (1000–9999).
h	Displays the hour without a leading 0 (0–24).
hh	Displays the hour with a leading 0 (00–24).
n	Displays the minute without a leading 0 (0–59).
nn	Displays the minute with a leading 0 (00–59).
s	Displays the second without a leading 0 (0–59).
ss	Displays the second with a leading 0 (00–59).
ttttt	Displays the time. A leading 0 is displayed if specified by your locale.
AM/PM or AMPM	Displays the time using a 12-hour clock. Displays an uppercase AM for time values before 12 noon. Displays an uppercase PM for time values after 12 noon and before 12 midnight.
am/pm	Displays the time using a 12-hour clock. Displays a lowercase am or pm at the end.
A/P	Displays the time using a 12-hour clock. Displays an uppercase A or P at the end.
a/p	Displays the time using a 12-hour clock. Displays a lowercase a or p at the end.

Example

```
Const crlf = Chr$(13) + Chr$(10)
```

```
Sub Main
  a# = 1199.234
  mesg = "Some general formats for '" & a# & "' are:"
  mesg = mesg & Format$(a#,"General Number") & crlf
  mesg = mesg & Format$(a#,"Currency") & crlf
  mesg = mesg & Format$(a#,"Standard") & crlf
  mesg = mesg & Format$(a#,"Fixed") & crlf
  mesg = mesg & Format$(a#,"Percent") & crlf
  mesg = mesg & Format$(a#,"Scientific") & crlf
  mesg = mesg & Format$(True,"Yes/No") & crlf
  mesg = mesg & Format$(True,"True/False") & crlf
  mesg = mesg & Format$(True,"On/Off") & crlf
  mesg = mesg & Format$(a#,"0,0.00") & crlf
  mesg = mesg & Format$(a#,"##,###,###.###") & crlf
  Session.Echo mesg
  da$ = Date$
  mesg = "Some date formats for '" & da$ & "' are:"
  mesg = mesg & Format$(da$,"General Date") & crlf
  mesg = mesg & Format$(da$,"Long Date") & crlf
  mesg = mesg & Format$(da$,"Medium Date") & crlf
  mesg = mesg & Format$(da$,"Short Date") & crlf
  Session.Echo mesg
  ti$ = Time$
  mesg = "Some time formats for '" & ti$ & "' are:"
  mesg = mesg & Format$(ti$,"Long Time") & crlf
  mesg = mesg & Format$(ti$,"Medium Time") & crlf
  mesg = mesg & Format$(ti$,"Short Time") & crlf
  Session.Echo mesg
End Sub
```

See Also Character and String Manipulation on page 3

FreeFile

Syntax `FreeFile [([rangenumber])]`

Description Returns an **`Integer`** containing the next available file number. This function returns the next available file number within the specified range. If **`rangenumber`** is 0, then a number between 1 and 255 is returned; if 1, then a number between 256 and 511 is returned. If **`rangenumber`** is not specified, then a number between 1 and 255 is returned.

The function returns 0 if there is no available file number in the specified range.

The number returned is suitable for use in the **`Open`** statement.

Example

```
Sub Main
  a = FreeFile
  Session.Echo "The next free file number is: " & a
End Sub
```

See Also Drive, Folder, and File Access on page 4

Function...End Function

Syntax

```
[Private | Public] [Static] Function name[(arglist)] [As ReturnType]
    [statements]
End Sub
```

where **`arglist`** is a comma-separated list of the following (up to 30 arguments are allowed):

```
[Optional] [ByVal | ByRef] parameter [()] [As type]
```

Description Creates a user-defined function. The **`Function`** statement has the following parts:

Part	Description
`Private`	Indicates that the function being defined cannot be called from other macros in other modules.
`Public`	Indicates that the function being defined can be called from other macros in other modules. If both the Private and Public keywords are missing, then Public is assumed.
`Static`	Recognized by the compiler but currently has no effect.
`name`	Name of the function, which must follow naming conventions: Must start with a letter. May contain letters, digits, and the underscore character (_). Punctuation and type-declaration characters are not allowed. The exclamation point (!) can appear within the name as long as it is not the last character, in which case it is interpreted as a type-declaration character. Must not exceed 80 characters in length. Additionally, the **`name`** parameter can end with an optional type-declaration character specifying the type of data returned by the function (i.e., any of the following characters: %, &, !, #, @).
`Optional`	Keyword indicating that the parameter is optional. All optional parameters must be of type variant. Furthermore, all parameters that follow the first optional parameter must also be optional. If this keyword is omitted, then the parameter is required. **Note:** You can use the **`IsMissing`** function to determine whether an optional parameter was actually passed by the caller.
`ByVal`	Keyword indicating that **`parameter`** is passed by value.
`ByRef`	Keyword indicating that **`parameter`** is passed by reference. If neither the ByVal nor the ByRef keyword is given, then ByRef is assumed.
`parameter`	Name of the parameter, which must follow the same naming conventions as those used by variables. This name can include a type-declaration character, appearing in place of **`As type`**.

Part	Description
`type`	Type of the parameter (integer, string, and so on). Arrays are indicated with parentheses. For example, an array of integers would be declared as follows: **`Function Test(a() As Integer)End Function`**
`ReturnType`	Type of data returned by the function. If the return type is not given, then variant is assumed. The **`ReturnType`** can only be specified if the function name (i.e., the **`name`** parameter) does not contain an explicit type-declaration character.

A function returns to the caller when either of the following statements is encountered: **`End Function`** or **`Exit Function`**.

Functions can be recursive.

Returning Values from Functions

To assign a return value, an expression must be assigned to the name of the function, as shown below:

```
Function TimesTwo(a As Integer) As Integer
  TimesTwo = a * 2
End Function
```

If no assignment is encountered before the function exits, then one of the following values is returned:

Value	Data Type Returned by the Function
0	Integer, long, single, double, currency
Zero-length string	String
Nothing	Object (or any data object)
Error	Variant
December 30, 1899	Date
False	Boolean

The type of the return value is determined by the **`As ReturnType`** clause in the **`Function`** statement itself. As an alternative, a type-declaration character can be added to the **`Function`** name. For example, the following two definitions of **`Test`** both return **`String`** values:

```
Function Test() As String
  Test = "Hello, world"
End Function
Function Test$()
  Test = "Hello, world"
End Function
```

Passing Parameters to Functions

Parameters are passed to a function either by value or by reference, depending on the declaration of that parameter in **`arglist`**. If the parameter is declared using the **`ByRef`** keyword, then any modifications to that passed parameter within the function change the value of that variable in the caller. If the parameter is declared using the **`ByVal`** keyword, then the value of that variable cannot be changed in the called function. If neither the **`ByRef`** or **`ByVal`** keywords are specified, then the parameter is passed by reference.

You can override passing a parameter by reference by enclosing that parameter within parentheses. For instance, the following example passes the variable j by reference, regardless of how the third parameter is declared in the **`arglist`** of **`UserFunction`**:

```
i = UserFunction(10,12,(j))
```

Optional Parameters

You can skip parameters when calling functions, as shown in the following example:

```
Function Test(a%,b%,c%) As Variant
End Function
Sub Main
  a = Test(1,,4)    'Parameter 2 was skipped.
End Sub
```

You can skip any parameter, with the following restrictions:

- The call cannot end with a comma. For instance, using the above example, the following is not valid:

  ```
  a = Test(1,,)
  ```

- The call must contain the minimum number of parameters as required by the called function. For instance, using the above example, the following are invalid:

  ```
  a = Test(,1)    'Only passes two out of three required
          'parameters.
  a = Test(1,2)    'Only passes two out of three required
          'parameters.
  ```

When you skip a parameter in this manner, the compiler creates a temporary variable and passes this variable instead. The value of this temporary variable depends on the data type of the corresponding parameter in the argument list of the called function, as described in the following table:

Value	Data Type
0	Integer, long, single, double, currency
Zero-length string	String
Nothing	Object (or any data object)

Value	Data Type
Error	Variant
December 30, 1899	Date
False	Boolean

Within the called function, you will be unable to determine whether a parameter was skipped unless the parameter was declared as a variant in the argument list of the function. In this case, you can use the **IsMissing** function to determine whether the parameter was skipped:

```
Function Test(a,b,c)
  If IsMissing(a) Or IsMissing(b) Then Exit Sub
End Function
```

Example

```
Function Factorial(n%) As Integer
  'This function calculates N! (N-factoral).
  f% = 1
  For i = n To 2 Step -1
    f = f * i
  Next i
  Factorial = f
End Function

Sub Main
    a% = 0
  Do While a% < 2
    a% = Val(InputBox$("Enter an integer number greater than 2.","Compute 
Factorial"))
  Loop
  b# = Factorial(a%)
  Session.Echo "The factoral of " & a% & " is: " & b#
End Sub
```

See Also Macro Control and Compilation on page 10

Fv

Syntax **Fv(rate, nper, pmt, pv, due)**

Description Calculates the future value of an annuity based on periodic fixed payments and a constant rate of interest. An annuity is a series of fixed payments made to an insurance company or other investment company over a period of time. Examples of annuities are mortgages and monthly savings plans. The **Fv** function requires the following named parameters:

Parameter	Description
rate	Double representing the interest rate per period. Make sure that annual rates are normalized for monthly periods (divided by 12).
nper	Double representing the total number of payments (periods) in the annuity.
pmt	Double representing the amount of each payment per period. Payments are entered as negative values, whereas receipts are entered as positive values.
pv	Double representing the present value of your annuity. In the case of a loan, the present value would be the amount of the loan, whereas in the case of a retirement annuity, the present value would be the amount of the fund.
due	Integer indicating when payments are due for each payment period. A 0 specifies payment at the end of each period, whereas a 1 indicates payment at the start of each period.

The **rate** and **nper** values must be expressed in the same units. If **rate** is expressed as a percentage per month, then **nper** must also be expressed in months. If **rate** is an annual rate, then the **nper** value must also be given in years.

Positive numbers represent cash received, whereas negative numbers represent cash paid out.

Example This example calculates the future value of 100 dollars paid periodically for a period of 10 years (120 months) at a rate of 10% per year (or .10/12 per month) with payments made on the first of the month. Note that payments are negative values.

```
Sub Main
  a# = Fv((.10/12),120,-100.00,0,1)
  Session.Echo "Future value is: " & Format(a#,"Currency")
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

G

Get

Syntax **`Get [#] filenumber, [recordnumber], variable`**

Description Retrieves data from a random or binary file and stores that data into the specified variable. The **`Get`** statement accepts the following parameters:

Parameter	Description
`filenumber`	Integer used to identify the file. This is the same number passed to the Open statement.
`recordnumber`	Long specifying which record is to be read from the file. For binary files, this number represents the first byte to be read starting with the beginning of the file (the first byte is 1). For random files, this number represents the record number starting with the beginning of the file (the first record is 1). This value ranges from 1 to 2147483647. If the **`recordnumber`** parameter is omitted, the next record is read from the file (if no records have been read yet, then the first record in the file is read). When this parameter is omitted, the commas must still appear, as in the following example: `Get #1,,recvar` **`If recordnumber`** is specified, and it overrides any previous change in file position specified with the Seek statement.
`variable`	Variable into which data will be read. The type of the variable determines how the data is read from the file, as described below.

With random files, a runtime error will occur if the length of the data being read exceeds the **`reclen`** parameter specified with the **`Open`** statement. If the length of the data being read is less than the record length, the file pointer is advanced to the start of the next record. With binary files, the data elements being read are contiguous; the file pointer is never advanced.

Variable types

The type of the **`variable`** parameter determines how data will be read from the file. It can be any of the following types:

Variable Type	File Storage Description
Integer	2 bytes are read from the file.
Long	4 bytes are read from the file.
String (variable-length)	In binary files, variable-length strings are read by first determining the specified string variable's length and then reading that many bytes from the file. For example, to read a string of eight characters: **`s$=String$(8,"")Get#1,,s$`** In random files, variable-length strings are read by first reading a 2-byte length and then reading that many characters from the file.
String (fixed-length)	Fixed-length strings are read by reading a fixed number of characters from the file equal to the string's declared length.
Double	8 bytes are read from the file (IEEE format).
Single	4 bytes are read from the file (IEEE format).
Date	8 bytes are read from the file (IEEE double format).
Boolean	2 bytes are read from the file. Nonzero values are True, and zero values are False.
Variant	A 2-byte VarType is read from the file, which determines the format of the data that follows. Once the VarType is known, the data is read individually, as described above. With user-defined errors, after the 2-byte VarType, a 2-byte unsigned integer is read and assigned as the value of the user-defined error, followed by 2 additional bytes of information about the error. The exception is with strings, which are always preceded by a 2-byte string length.
User-defined types	Each member of a user-defined data type is read individually. In binary files, variable-length strings within user-defined types are read by first reading a 2-byte length followed by the string's content. This storage is different from variable-length strings outside of user-defined types. When reading user-defined types, the record length must be greater than or equal to the combined size of each element within the data type.
Arrays	Arrays cannot be read from a file using the Get statement.
Object	Object variables cannot be read from a file using the Get statement.

Example

```
Sub Main
  Open "test.dat" For Random Access Write As #1
  For x = 1 to 10
```

```
    y% = x * 10
    Put #1,x,y
  Next x
  Close
  Open "test.dat" For Random Access Read As #1
  For y = 1 to 5
    Get #1,y,x%
    mesg = mesg & "Record " & y & ": " & x% & Basic.Eoln$
  Next y
  Session.Echo mesg
  Close
End Sub
```

See Also Drive, Folder, and File Access on page 4

GetAttr

Syntax `GetAttr(pathname)`

Description Returns an **`Integer`** containing the attributes of the specified file. The attribute value returned is the sum of the attributes set for the file. The value of each attribute is as follows:

Value	Constant	Includes
0	**`ebNormal`**	Read-only files, archive files, subdirectories, and files with no attributes
1	**`ebReadOnly`**	Read-only files
2	**`ebHidden`**	Hidden files
4	**`ebSystem`**	System files
9	**`ebVolume`**	Volume label
16	**`ebDirectory`**	Subdirectories
32	**`ebArchive`**	Files that have changed since the last backup
64	**`ebNone`**	Files with no attributes

To determine whether a particular attribute is set, you can **`And`** the values shown above with the value returned by **`GetAttr`**.

If the result is **`True`**, the attribute is set, as shown below:

```
Dim w As Integer
w = GetAttr("sample.txt")
If w And ebReadOnly Then Session.Echo "This file is read-only."
```

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  If Not FileExists("test.dat") Then
    Open "test.dat" For Random Access Write As #1
    Close
  End If
  y% = GetAttr("test.dat")
```

```
  If y% And ebNone Then mesg = mesg & _
    "No archive bit is set." & crlf
  If y% And ebReadOnly Then mesg = mesg & _
    "The read-only bit is set." & crlf
  If y% And ebHidden Then mesg = mesg & "The hidden bit is set." & _
    crlf
  If y% And ebSystem Then mesg = mesg & "The system bit is set." & _
    crlf
  If y% And ebVolume Then mesg = mesg & "Volume bit is set." &    crlf
  If y% And ebDirectory Then mesg = mesg & "Directory bit is set." &
  & crlf
  If y% And ebArchive Then mesg = mesg & "The archive bit is set."
  Session.Echo mesg
  Kill "test.dat"
End Sub
```

See Also Drive, Folder, and File Access on page 4

GetObject

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

Description Returns the object specified by **`pathname`** or returns a previously instantiated object of the given **`class`**. This function is used to retrieve an existing OLE Automation object, either one that comes from a file or one that has previously been instantiated.

The **`pathname`** argument specifies the full pathname of the file containing the object to be activated. The application associated with the file is determined by OLE at runtime. For example, suppose that a file called c:\docs\resume.doc was created by a word processor called wordproc.exe. The following statement would invoke wordproc.exe, load the file called c:\docs\resume.doc, and assign that object to a variable:

```
Dim doc As Object
Set doc = GetObject("c:\docs\resume.doc")
```

To activate a part of an object, add an exclamation point to the filename followed by a string representing the part of the object that you want to activate. For example, to activate the first three pages of the document in the previous example:

```
Dim doc As Object
Set doc = GetObject("c:\docs\resume.doc!P1-P3")
```

The **`GetObject`** function behaves differently depending on whether the first named parameter is omitted. The following table summarizes the different behaviors of **`GetObject`**:

Pathname	Class	`GetObject` Returns
Not specified	Specified	A reference to an existing instance of the specified object. A runtime error results if the object is not already loaded.
" "	Specified	A reference to a new object (as specified by class). A runtime error occurs if an object of the specified class cannot be found. This is the same as **`CreateObject`**.
Specified	Not specified	The default object from pathname. The application to activate is determined by OLE based on the given filename.
Specified	Specified	The object given class from the file given by pathname. A runtime error occurs if an object of the given class cannot be found in the given file.

Examples This first example instantiates the existing copy of Excel.

```
Dim Excel As Object
Set Excel = GetObject(,"Excel.Application")
```

This second example loads the OLE server associated with a document.

```
Dim MyObject As Object
Set MyObject = GetObject("c:\documents\resume.doc",)
```

See Also Objects on page 18; DDE Access on page 19

GoSub

Syntax **`GoSub label`**

Description Causes execution to continue at the specified label. Execution can later be returned to the statement following the **`GoSub`** by using the **`Return`** statement. The **`label`** parameter must be a label within the current function or subroutine. **`GoSub`** outside the context of the current function or subroutine is not allowed.

Example

```
Sub Main
  uname$ = Ucase$(InputBox$("Enter your name:","Enter Name"))
  GoSub CheckName
  Session.Echo "Hello, " & uname$
  Exit Sub
CheckName:
  If (uname$ = "") Then
    GoSub BlankName
  ElseIf uname$ = "MICHAEL" Then
    GoSub RightName
  Else
    GoSub OtherName
  End If
  Return
BlankName:
  Session.Echo "No name? Clicked Cancel? I'm shutting down."
```

```
  Exit Sub
RightName:
  Return
OtherName:
  Session.Echo "I am renaming you MICHAEL!"
  uname$ = "MICHAEL"
  Return
End Sub
```

See Also Macro Control and Compilation on page 10

Goto

Syntax `Goto label`

Description Transfers execution to the line containing the specified label. The compiler will produce an error if **`label`** does not exist. The **`label`** must appear within the same subroutine or function as the **`Goto`**.

Labels are identifiers that follow these rules:

- Must begin with a letter.
- May contain letters, digits, and the underscore character.
- Must not exceed 80 characters in length.
- Must be followed by a colon (:).

Labels are not case-sensitive.

When you're running a macro within the macro editor, you can break out of an infinite loop by pressing Ctrl+Break.

Example

```
Sub Main
  uname$ = Ucase$(InputBox$("Enter your name:","Enter Name"))
  If uname$ = "MICHAEL" Then
    Goto RightName
  Else
    Goto WrongName
  End If
WrongName:
  If (uname$ = "") Then
    Session.Echo "No name? Clicked Cancel? I'm shutting down."
  Else
    Session.Echo "I am renaming you MICHAEL!"
    uname$ = "MICHAEL"
    Goto RightName
  End If
  Exit Sub
RightName:
  Session.Echo "Hello, MICHAEL!"
End Sub
```

See Also Macro Control and Compilation on page 10

GroupBox

Syntax `GroupBox x,y,width,height,title$ [,.Identifier]`

Description Defines a group box within a dialog template. This statement can only appear within a dialog template (i.e., between the **`Begin Dialog`** and **`End Dialog`** statements).

The group box control is used for static display only the user cannot interact with a group box control.

Separator lines can be created using group box controls. This is accomplished by creating a group box that is wider than the width of the dialog and extends below the bottom of the dialog; i.e., three sides of the group box are not visible.

If **`title$`** is a zero-length string, then the group box is drawn as a solid rectangle with no title.

The **`GroupBox`** statement requires the following parameters:

Parameter	Description
`x, y`	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
`width`, **`height`**	Integer coordinates specifying the dimensions of the control in dialog units.
`title$`	String containing the label of the group box. If **`title$`** is a zero-length string, then no title will appear.
`.Identifier`	Optional parameter that specifies the name by which this control can be referenced by statements in a dialog function (such as **`DlgFocus`** and **`DlgEnable`**). If omitted, then the first two words of **`title$`** are used.

Example

```
Sub Main
  Begin Dialog OptionsTemplate 16,32,128,84,"Options"
    GroupBox 4,4,116,40,"Window Options"
    CheckBox 12,16,60,8,"Show &Toolbar",.ShowToolbar
    CheckBox 12,28,68,8,"Show &Status Bar",.ShowStatusBar
    GroupBox -12,52,152,48," ",.SeparatorLine
    OKButton 16,64,40,14,.OK
    CancelButton 68,64,40,14,.Cancel
  End Dialog
  Dim OptionsDialog As OptionsTemplate
  Dialog OptionsDialog
End Sub
```

See Also User Interaction on page 16

H

HelpButton

Syntax **`HelpButton x,y,width,height,HelpFileName$,HelpContext, [,.Identifier]`**

Description Defines a help button within a dialog template. This statement can only appear within a dialog template (i.e., between the **`Begin Dialog`** and **`End Dialog`** statements). The **`HelpButton`** statement takes the following parameters:

Parameter	Description
`x,y`	Integer position of the control (in dialog units) relative to the upper left corner of the dialog.
`width,height`	Integer dimensions of the control in dialog units.
`HelpFileName$`	String expression specifying the name of the help file to be invoked when the button is selected.
`HelpContext`	Long expression specifying the ID of the topic within **`HelpFileName$`** containing context-sensitive help.
`.Identifier`	Name by which this control can be referenced by statements in a dialog function (such as **`DlgFocus`** and **`DlgEnable`**).

When the user selects a help button, the associated help file is located at the indicated topic. Selecting a help button does not remove the dialog. Similarly, no actions are sent to the dialog procedure when a help button is selected.

When a help button is present within a dialog, it can be automatically selected by pressing the help key F1.

Example

```
Sub Main
  Begin Dialog HelpDialogTemplate ,,180,96,"Untitled"
    OKButton 132,8,40,14
    CancelButton 132,28,40,14
    HelpButton 132,48,40,14,"", 10
    Text 16,12,88,12,"Please click ""Help"".",.Text1
  End Dialog
Dim HelpDialog As HelpDialogTemplate
Dialog HelpDialog
End Sub
```

See Also User Interaction on page 16

Hex, Hex$

Syntax `Hex[$](number)`

Description Returns a **`String`** containing the hexadecimal equivalent of **`number`**. **`Hex$`** returns a **`String`**, whereas **`Hex`** returns a **`String`** variant. The returned string contains only the number of hexadecimal digits necessary to represent the number, up to a maximum of eight.

The **`number`** parameter can be any type but is rounded to the nearest whole number before converting to hex. If the passed number is an integer, then a maximum of four digits are returned; otherwise, up to eight digits can be returned.

The **`number`** parameter can be any expression convertible to a number. If **`number`** is **`Null`**, then **`Null`** is returned. **`Empty`** is treated as 0.

Example

```
Sub Main
  Do
    xs$ = InputBox$("Enter a number to convert:","Hex Convert")
    x = Val(xs$)
    If x <> 0 Then
      Session.Echo "Dec: " & x & "  Hex: " & Hex$(x)
    Else
      Session.Echo "Goodbye."
    End If
  Loop While x <> 0
End Sub
```

See Also Character and String Manipulation on page 3

Hour

Syntax `Hour(time)`

Description Returns the hour of the day encoded in the specified **`time`** parameter. The value returned is an **`Integer`** between 0 and 23 inclusive. The **`time`** parameter is any expression that converts to a **`Date`**.

Example

```
Sub Main
  xt# = TimeValue(Time$())
  xh# = Hour(xt#)
  xm# = Minute(xt#)
  xs# = Second(xt#)
  Session.Echo "The current time is: " & xh# & ":" & xm# & ":" & xs#
End Sub
```

See Also Time and Date Access on page 17

I

If...Then...Else

Syntax 1 `If condition Then statements [Else else_statements]`

Syntax 2

```
If condition Then
  [statements]
[ElseIf else_condition Then
  [elseif_statements]]
[Else
  [else_statements]]
End If
```

Description Conditionally executes a statement or group of statements. The single-line conditional statement (syntax 1) has the following parameters:

Parameter	Description
`condition`	Any expression evaluating to a boolean value.
`Statements`	One or more statements separated with colons. This group of statements is executed when **`condition`** is True.
`else_statements`	One or more statements separated with colons. This group of statements is executed when **`condition`** is False.

The multiline conditional statement (syntax 2) has the following parameters:

Parameter	Description
`condition`	Any expression evaluating to a boolean value.
`Statements`	One or more statements to be executed when **`condition`** is True.

Parameter	Description
else_condition	Any expression evaluating to a boolean value. The **else_condition** is evaluated if **condition** is False.
elseif_statements	One or more statements to be executed when **condition** is False and **else_condition** is True.
else_statments	One or more statements to be executed when both **condition** and **else_condition** are False.

There can be as many **ElseIf** conditions as required.

Example

```
Sub Main
  uname$ = Ucase$(InputBox$("Enter your name:","Enter Name"))
  If uname$ = "MICHAEL" Then GoSub MikeName
  If uname$ = "MIKE" Then
    GoSub MikeName
    Exit Sub
  End If
  If uname$ = "" Then
    Session.Echo "Since you don't have a name, I'll call you MIKE!"
    uname$ = "MIKE"
    GoSub MikeName
  ElseIf uname$ = "MICHAEL" Then
    GoSub MikeName
  Else
    GoSub OtherName
  End If
  Exit Sub
MikeName:
  Session.Echo "Hello, MICHAEL!"
  Return
OtherName:
  Session.Echo "Hello, " & uname$ & "!"
  Return
End Sub
```

See Also Macro Control and Compilation on page 10

Iif

Syntax **Iif(expression, truepart, falsepart)**

Description Returns **truepart** if condition is **True**; otherwise, returns **falsepart**. Both expressions are calculated before **Iif** returns. The **Iif** function is shorthand for the following construct:

```
If condition Then
  variable = truepart
Else
  variable = falsepart
End If
```

Example

```
Sub Main
  s$ = "Car"
  Session.Echo Iif(s$ = "Car","Nice Car","Nice Automobile")
End Sub
```

See Also Macro Control and Compilation on page 10

IMEStatus

Syntax `IMEStatus[()]`

Description Returns the current status of the input method editor. The **`IMEStatus`** function returns one of the following constants for Japanese locales:

Constant	Value	Description
`ebIMENoOp`	0	IME not installed.
`EbIMEOn`	1	IME on.
`EbIMEOff`	2	IME off.
`EbIMEDisabled`	3	IME disabled.
`EbIMEHiragana`	4	Hiragana double-byte character.
`EbIMEKatakanaDbl`	5	Katakana double-byte characters.
`EbIMEKatakanaSng`	6	Katakana single-byte characters.
`EbIMEAlphaDbl`	7	Alphanumeric double-byte characters.
`EbIMEAlphaSng`	8	Alphanumeric single-byte characters.

For Chinese locales, one of the following constants are returned:

Constant	Value	Description
`ebIMENoOp`	0	IME not installed.
`EbIMEOn`	1	IME on.
`EbIMEOff`	2	IME off.

For Korean locales, this function returns a value with the first 5 bits having the following meaning:

Bit	If Not Set (Or 0)	If Set (Or 1)
Bit 0	IME not installed	IME installed
Bit 1	IME disabled	IME enabled
Bit 2	English mode	Hangeul mode
Bit 3	Banja mode (single-byte)	Junja mode (double-byte)
Bit 4	Normal mode	Hanja conversion mode

Note You can test for the different bits using the **And** operator as follows:

```
a = IMEStatus()
If a And 1 Then …  'Test for bit 0
If a And 2 Then …  'Test for bit 1
If a And 4 Then …  'Test for bit 2
If a And 8 Then …  'Test for bit 3
If a And 16 Then …  'Test for bit 4
```

This function always returns 0 if no input method editor is installed.

Example

```
Sub Main
  a = IMEStatus()
  Select case a
  Case 0
    Session.Echo "IME not installed."
  Case 1
    Session.Echo "IME on."
  Case 2
    Session.Echo "IME off."
  End Select
End Sub
```

See Also Operating System Control on page 15

Imp (operator)

Syntax `result = expression1 Imp expression2`

Description Performs a logical or binary implication on two expressions. If both expressions are either **Boolean**, **Boolean** variants, or **Null** variants, then a logical implication is performed as follows:

Expression One	Expression Two	Result
True	True	True
True	False	False
True	Null	Null
False	True	True
False	False	True
False	Null	True
Null	True	True
Null	False	Null
Null	Null	Null

Binary implication

If the two expressions are **`Integer`**, then a binary implication is performed, returning an **`Integer`** result. All other numeric types (including **`Empty`** variants) are converted to **`Long`** and a binary implication is then performed, returning a **`Long`** result.

Binary implication forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions, according to the following table:

Bit in Expression One	Bit in Expression Two	Result
1	1	1
0	1	1
1	0	0
0	0	1

Example

```
Sub Main
  a = 10 : b = 20 : c = 30 : d = 40
  If (a < b) Imp (c < d) Then
    Session.Echo "a is less than b implies that c is less than d."
  Else
    Session.Echo "a is less than b does not imply that c is less than d."
  End If
  If (a < b) Imp (c > d) Then
    Session.Echo "a is less than b implies that c is greater than d."
  Else
    Session.Echo "a is less than b does not imply that c is greater than d."
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Input#

Syntax **`Input [#]filenumber%,variable[,variable]…`**

Description Reads data from the file referenced by **`filenumber`** into the given variables. Each **`variable`** must be type-matched to the data in the file. For example, a **`String`** variable must be matched to a string in the file. The following parsing rules are observed while reading each variable in the variable list:

- Leading white space is ignored (spaces and tabs).
- When reading **`String`** variables, if the first character on the line is a quotation mark, then characters are read up to the next quotation mark or the end of the line, whichever comes first. Blank lines are read as empty strings. If the first character read is not a quotation mark, then characters are read up to the first comma or the end of the line, whichever comes first. String delimiters (quotes, comma, end-of-line) are not included in the returned string.

- When reading numeric variables, scanning of the number stops when the first non-numeric character (such as a comma, a letter, or any other unexpected character) is encountered. Numeric errors are ignored while reading numbers from a file. The resultant number is automatically converted to the same type as the variable into which the value will be placed. If there is an error in conversion, then 0 is stored into the variable.
- After reading the number, input is skipped up to the next delimiter—a comma, an end-of-line, or an end-of-file.
- Numbers must adhere to any of the following syntax:

```
[-|+]digits[.digits][E[-|+]digits][!|#|%|&|@]
&Hhexdigits[!|#|%|&]
&[O]octaldigits[!|#|%|&|@]
```

- When reading **Boolean** variables, the first character must be #; otherwise, a runtime error occurs. If the first character is #, then input is scanned up to the next delimiter (a comma, an end-of-line, or an end-of-file). If the input matches #FALSE#, then **False** is stored in the **Boolean**; otherwise, **True** is stored.
- When reading date variables, the first character must be #; otherwise, a runtime error occurs. If the first character is #, then the input is scanned up to the next delimiter (a comma, an end-of-line, or an end-of-file). If the input ends in a # and the text between the #'s can be correctly interpreted as a date, then the date is stored; otherwise, December 31, 1899, is stored.

Normally, dates that follow the universal date format are input from sequential files. These dates use this syntax:

```
#YYYY-MM-DD HH:MM:SS#
```

where **YYYY** is a year between 100 and 9999, **MM** is a month between 1 and 12, **DD** is a day between 1 and 31, **HH** is an hour between 0 and 23, **MM** is a minute between 0 and 59, and **SS** is a second between 0 and 59.

- When reading **Variant** variables, if the data begins with a quotation mark, then a string is read consisting of the characters between the opening quotation mark and the closing quotation mark, end-of-line, or end-of-file.

If the input does not begin with a quotation mark, then input is scanned up to the next comma, end-of-line, or end-of-file and a determination is made as to what data is being represented. If the data cannot be represented as a number, **Date**, **Error**, **Boolean**, or **Null**, then it is read as a string.

The following table describes how special data is interpreted as variants:

Special Data	Interpreted as Variant
Blank line	Read as an empty variant.
#NULL#	Read as a null variant.
TRUE#	Read as a boolean variant.

Special Data	Interpreted as Variant
`#FALSE#`	Read as a boolean variant.
`ERROR code#`	Read as a user-defined error.
`Date#`	Read as a date variant.
`"text"`	Read as a string variant.

- If an error occurs in interpretation of the data as a particular type, then that data is read as a **`String`** variant.
- When reading numbers into variants, the optional type-declaration character determines the VarType of the resulting variant. If no type-declaration character is specified, then the compiler will read the number according to the following rules:
  - **Rule 1:** If the number contains a decimal point or an exponent, then the number is read as **`Currency`**. If there is an error converting to **`Currency`**, then the number is treated as a **`Double`**.
  - **Rule 2:** If the number does not contain a decimal point or an exponent, then the number is stored in the smallest of the following data types that most accurately represents that value: integer, long, currency, double.
- End-of-line is interpreted as either a single line feed, a single carriage return, or a carriage-return/line-feed pair. Thus, text files from any platform can be interpreted using this command.
- The **`filenumber`** parameter is a number that is used to refer to the open file the number passed to the **`Open`** statement.
- The **`filenumber`** must reference a file opened in **`Input`** mode. It is good practice to use the **`Write`** statement to write date elements to files read with the **`Input`** statement to ensure that the variable list is consistent between the input and output routines.
- Null characters are ignored.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Open "test.dat" For Output As #1
  Write #1,2112,"David","McCue","123-45-6789"
  Close
  Open "test.dat" For Input As #1
  Input #1,x%,st1$,st2$,st3$
  mesg = "Employee " & x% & " Information" & crlf & crlf
  mesg = mesg & "First Name: " & st1$ & crlf
  mesg = mesg & "Last Name: "& st2$ & crlf
  mesg = mesg & "Social Security Number: " & sy3$
  Session.Echo mesg
  Close
  Kill "test.dat"
End Sub
```

See Also Drive, Folder, and File Access on page 4

Input, Input$, InputB, InputB$

Syntax `Input[$](numchars,[#]filenumber)`
`InputB[$](numbytes,[#]filenumber)`

Description Returns a specified number of characters or bytes read from a given sequential file. The **`Input$`** and **`InputB$`** functions return a **`String`**, whereas **`Input`** and **`InputB`** return a **`String`** variant. The following parameters are required:

Parameter	Description
`numchars`	Integer containing the number of characters to be read from the file.
`numbytes`	Integer containing the number of bytes to be read from the file.
`filenumber`	Integer referencing a file opened in either Input or Binary mode. This is the same number passed to the Open statement.

The **`Input`** and **`Input$`** functions read all characters, including spaces and end-of-lines. Null characters are ignored.

The InputB and InputB$ functions are used to read byte data from a file.

Example

```
Const crlf = Chr$(13) & Chr$(10)

Sub Main
  x& = FileLen("c:\autoexec.bat")
  If x& > 0 Then
    Open "c:\autoexec.bat" For Input As #1
  Else
    Session.Echo "File not found or empty."
    Exit Sub
  End If
  If x& > 80 Then
    ins = Input(80,#1)
  Else
    ins = Input(x,#1)
  End If
  Close
  Session.Echo "File length: " & x& & crlf & ins
End Sub
```

See Also Drive, Folder, and File Access on page 4

InputBox, InputBox$

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

Description Displays a dialog with a text box into which the user can type. The content of the text box is returned as a **`String`** (in the case of **`InputBox$`**) or as a **`String`** variant (in the case of **`InputBox`**). A zero-length string is returned if the user selects Cancel. The **InputBox/InputBox$** functions take the following named parameters:

Parameter	Description
prompt	Text to be displayed above the text box. The **prompt** parameter can contain multiple lines, each separated with an end-of-line (a carriage return, line feed, or carriage-return/line-feed pair). A runtime error is generated if **prompt** is null.
title	Caption of the dialog. If this parameter is omitted, then no title appears as the dialog's caption. A runtime error is generated if **title** is null.
default	Default response. This string is initially displayed in the text box. A runtime error is generated if **default** is null.
xpos, **ypos**	Integer coordinates, given in twips (twentieths of a point), specifying the upper left corner of the dialog relative to the upper left corner of the screen. If the position is omitted, then the dialog is positioned on or near the application executing the macro.
helpfile	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then **context** must also be specified.
context	Number specifying the ID of the topic within **helpfile** for this dialog's help. If this parameter is specified, then **helpfile** must also be specified.

You can type a maximum of 255 characters into **InputBox**.

If both the **helpfile** and **context** parameters are specified, then a Help button is added in addition to the OK and Cancel buttons. Context-sensitive help can be invoked by selecting this button or using the help key F1. Invoking help does not remove the dialog.

When Cancel is selected, an empty string is returned. An empty string is also returned when the user selects the OK button with no text in the input box. Thus, it is not possible to determine the difference between these two situations. If you need to determine the difference, you should create a user-defined dialog or use the AskBox function.

Example

```
Sub Main
  s$ = InputBox$("File to copy:","Copy","sample.txt")
End Sub
```

See Also User Interaction on page 16

InStr, InstrB

Syntax

```
InStr([start,] search, find [,compare])
InStrB([start,] search, find [,compare])
```

Description Returns the first character position of string **find** within string **search**. The **InStr** function takes the following parameters:

Parameter	Description
`start`	Integer specifying the character position (for Instr) or byte position (for InstrB) where searching begins. The **`start`** parameter must be between 1 and 32767. If this parameter is omitted, then the search starts at the beginning (**`start`** = 1).
`search`	Text to search. This can be any expression convertible to a string.
`find`	Text for which to search. This can be any expression convertible to a string.
`compare`	Integer controlling how string comparisons are performed. It can be any of the following values: 0 String comparisons are case-sensitive. 1 String comparisons are case-insensitive. Any other value produces a runtime error. If this parameter is omitted, then string comparisons use the current Option Compare setting. If no Option Compare statement has been encountered, then Binary is used (i.e., string comparisons are case-sensitive).

If the string is found, then its character position within **`search`** is returned, with 1 being the character position of the first character.

The InStr and InStrB functions observe the following additional rules:

- If either **`search`** or **`find`** is **`Null`**, then **`Null`** is returned.
- If the **`compare`** parameter is specified, then **`start`** must also be specified. In other words, if there are three parameters, then it is assumed that these parameters correspond to **`start`**, **`search`**, and **`find`**.
- A runtime error is generated if **`start`** is null.
- A runtime error is generated if **`compare`** is not 0 or 1.
- If **`search`** is empty, then 0 is returned.
- If **`find`** is empty, then **`start`** is returned. If **`start`** is greater than the length of **`search`**, then 0 is returned.
- A runtime error is generated if **`start`** is less than or equal to zero.

The **`InStr`** and **`InStrB`** functions operate on character and byte data respectively. The Instr function interprets the **`start`** parameter as a character, performs a textual comparisons, and returns a character position. The **`InStrB`** function, on the other hand, interprets the **`start`** parameter as a byte position, performs binary comparisons, and returns a byte position.

On SBCS platforms, the **`InStr`** and **`InStrB`** functions are identical.

Example

```
Sub Main
  a$ = "This string contains the name Stuart and other characters."
  x% = InStr(a$,"Stuart",1)
  If x% <> 0 Then
    b$ = Mid$(a$,x%,6)
    Session.Echo b$ & " was found."
    Exit Sub
  Else
    Session.Echo "Stuart not found."
  End If
End Sub
```

See Also Character and String Manipulation on page 3

Int

Syntax `Int(number)`

Description Returns the integer part of **`number`**. This function returns the integer part of a given value by returning the first integer less than the **`number`**. The sign is preserved. The Int function returns the same type as **`number,`** with the following exceptions:

- If **`number`** is **`Empty`**, then an **`Integer`** variant of value 0 is returned.
- If **`number`** is a string, then a double variant is returned.
- If **`number`** is null, then a null variant is returned.

Example

```
Sub Main
  a# = -1234.5224
  b% = Int(a#)
  Session.Echo "The integer part of -1234.5224 is: " & b%
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

Integer (data type)

Syntax `Integer`

Description Used to declare whole numbers with up to four digits of precision. **`Integer`** variables are used to hold numbers within the following range:

```
–32768 <= integer <= 32767
```

Internally, integers are 2-byte short values. Thus, when appearing within a structure, integers require 2 bytes of storage. When used with binary or random files, 2 bytes of storage are required.

When passed to external routines, integer values are sign-extended to the size of an integer on that platform (either 16 or 32 bits) before pushing onto the stack.

The type-declaration character for integer is %.

See Also Keywords, Data Types, Operators, and Expressions on page 6

IPmt

Syntax `IPmt(rate, per, nper, pv, fv, due)`

Description Returns the interest payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate. An annuity is a series of fixed payments made to an insurance company or other investment company over a period of time. Examples of annuities are mortgages, monthly savings plans, and retirement plans. The following table describes the named parameters:

Parameter	Description
`rate`	Double representing the interest rate per period. If the payment periods are monthly, be sure to divide the annual interest rate by 12 to get the monthly rate.
`per`	Double representing the payment period for which you are calculating the interest payment. If you want to know the interest paid or received during period 20 of an annuity, this value would be 20.
`nper`	Double representing the total number of payments in the annuity. This is usually expressed in months, and you should be sure that the interest rate given above is for the same period that you enter here.
`pv`	Double representing the present value of your annuity. In the case of a loan, the present value would be the amount of the loan because that is the amount of cash you have in the present. In the case of a retirement plan, this value would be the current value of the fund because you have a set amount of principal in the plan.
`fv`	Double representing the future value of your annuity. In the case of a loan, the future value would be zero because you will have paid it off. In the case of a savings plan, the future value would be the balance of the account after all payments are made.
`due`	Integer indicating when payments are due. If this parameter is 0, then payments are due at the end of each period (usually, the end of the month). If this value is 1, then payments are due at the start of each period (the beginning of the month).

The **`rate`** and **`nper`** parameters must be expressed in the same units. If **`rate`** is expressed in percentage paid per month, then **`nper`** must also be expressed in months. If **`rate`** is an annual rate, then the period given in **`nper`** should also be in years or the annual **`rate`** should be divided by 12 to obtain a monthly rate.

If the function returns a negative value, it represents interest you are paying out, whereas a positive value represents interest paid to you.

Example This example calculates the amount of interest paid on a $1,000.00 loan financed over 36 months with an annual interest rate of 10%. Payments are due at the beginning of the month. The interest paid during the first 10 months is displayed in a table.

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  For x = 1 to 10
    ipm# = IPmt((.10/12),x,36,1000,0,1)
    mesg = mesg & Format(x,"00") & " : " & Format(ipm#," 0,0.00") & crlf
  Next x
  Session.Echo mesg
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

IRR

Syntax `IRR(valuearray(),guess)`

Description Returns the internal rate of return for a series of periodic payments and receipts. The internal rate of return is the equivalent rate of interest for an investment consisting of a series of positive and/or negative cash flows over a period of regular intervals. It is usually used to project the rate of return on a business investment that requires a capital investment up front and a series of investments and returns on investment over time. The **IRR** function requires the following named parameters:

Parameter	Description
valuearray()	Array of double numbers that represent payments and receipts. Positive values are payments, and negative values are receipts.
	There must be at least one positive and one negative value to indicate the initial investment (negative value) and the amount earned by the investment (positive value).
guess	Double containing your guess as to the value that the **IRR** function will return. The most common guess is .1 (10 percent).

The value of **IRR** is found by iteration. It starts with the value of **guess** and cycles through the calculation adjusting **guess** until the result is accurate within 0.00001 percent. After 20 tries, if a result cannot be found, **IRR** fails, and the user must pick a better guess.

Example This example illustrates the purchase of a lemonade stand for $800 and a series of incomes from the sale of lemonade over 12 months. The projected incomes for this example are generated in two For...Next Loops, and then the internal rate of return is calculated and displayed. (Not a bad investment!)

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
```

```
  Dim valu#(12)
  valu(1) = -800            'Initial investment
  mesg = valu#(1) & ", "
  'Calculate the second through fifth months' sales.
  For x = 2 To 5
    valu(x) = 100 + (x * 2)
    mesg = mesg & valu(x) & ", "
  Next x
  'Calculate the sixth through twelfth months' sales.
  For x = 6 To 12
    valu(x) = 100 + (x * 10)
    mesg = mesg & valu(x) & ", "
  Next x
  'Calculate the equivalent investment return rate.
  retrn# = IRR(valu,.1)
  mesg = "The values: " & crlf & mesg & crlf & crlf
  Session.Echo mesg & "Return rate: " & Format(retrn#,"Percent")
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

Is

Syntax **`object Is [object | Nothing]`**

Description Returns **`True`** if the two operands refer to the same object; returns **`False`** otherwise. This operator is used to determine whether two object variables refer to the same object. Both operands must be object variables of the same type (i.e., the same data object type or both of type **`Object`**).

The **`Nothing`** constant can be used to determine whether an object variable is uninitialized:

```
If MyObject Is Nothing Then Session.Echo "MyObject is uninitialized."
```

Uninitialized object variables reference no object.

When comparing OLE Automation objects, the **`Is`** operator will only return **`True`** if the operands reference the same OLE Automation object. This is different from data objects. For example, the following use of **`Is`** (using the object class called **`excel.application`**) returns **`True`**:

```
Dim a As Object
Dim b As Object
a = CreateObject("excel.application")
b = a
If a Is b Then Beep
```

The following use of **`Is`** will return **`False`**, even though the actual objects may be the same:

```
Dim a As Object
Dim b As Object
a = CreateObject("excel.application")
b = GetObject(,"excel.application")
If a Is b Then Beep
```

The **Is** operator may return **False** in the above case because, even though a and b reference the same object, they may be treated as different objects by OLE 2.0 (this is dependent on the OLE 2.0 server application).

Example

```
Sub Main
  Dim CurrentSession As Object
  Set CurrentSession = Application.ActiveSession
  If CurrentSession.Circuit = Nothing Then
    MsgBox "No communications method selected."
  End If
End

Sub InsertDate(ByVal WinWord As Object)
  If WinWord Is Nothing Then
    Session.Echo "Object variant is not set."
  Else
    WinWord.Insert Date$
  End If
End Sub

Sub Main
  Dim WinWord As Object
  On Error Resume Next
  WinWord = CreateObject("word.basic")
  InsertDate WinWord
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Objects on page 18

IsDate

Syntax **IsDate(expression)**

Description Returns **True** if **expression** can be legally converted to a date; returns **False** otherwise.

Example

```
Sub Main
  Dim a As Variant
Retry:
  a = InputBox("Enter a date.", "Enter Date")
  If IsDate(a) Then
    Session.Echo Format(a,"long date")
  Else
    Session.Echo "Not quite, please try again!"
    Goto Retry
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Time and Date Access on page 17

IsEmpty

Syntax `IsEmpty(expression)`

Description Returns **`True`** if **`expression`** is a **`Variant`** variable that has never been initialized; returns **`False`** otherwise. The **`IsEmpty`** function is the same as the following:

```
(VarType(expression) = ebEmpty)
```

Example

```
Sub Main
  Dim a As Variant
  If IsEmpty(a) Then
    a = 1.0#        'Give uninitialized data a Double value 0.0.
    Session.Echo "The variable has been initialized to: " & a
  Else
    Session.Echo "The variable was already initialized!"
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

IsError

Syntax `IsError(expression)`

Description Returns **`True`** if expression is a user-defined error value; returns **`False`** otherwise.

Example

```
Function Div(ByVal a,ByVal b) As Variant
  If b = 0 Then
    Div = CVErr(2112)    'Return a special error value.
  Else
    Div = a / b      'Return the division.
  End If
End Function

Sub Main
  Dim a As Variant
  a = Div(10,12)
  If IsError(a) Then
    Session.Echo "The following error occurred: " & CStr(a)
  Else
    Session.Echo "The result is: " & a
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

IsMissing

Syntax `IsMissing(argname)`

Description Returns **`True`** if **`argname`** was passed to the current subroutine or function; returns **`False`** if omitted. The **`IsMissing`** function is used with variant variables passed as optional parameters (using the **`Optional`** keyword) to the current subroutine or function. For nonvariant variables or variables that were not declared with the **`Optional`** keyword, **`IsMissing`** will always return **`True`**.

Example

```
Sub Test(AppName As String,Optional isMinimize As Variant)
  app = Shell(AppName)
  If Not IsMissing(isMinimize) Then
    AppMinimize app
  Else
    AppMaximize app
  End If
End Sub

Sub Main
  Test "Notepad"      'Maximize this application
  Test "Notepad",True    'Minimize this application
End Sub
```

See Also Macro Control and Compilation on page 10

IsNull

Syntax **`IsNull(expression)`**

Description Returns **`True`** if **`expression`** is a **`Variant`** variable that contains no valid data; returns **`False`** otherwise. The **`IsNull`** function is the same as the following:

```
(VarType(expression) = ebNull)
```

Example

```
Sub Main
  Dim a As Variant    'Initialized as Empty
  If IsNull(a) Then Session.Echo "The variable contains no valid data."
  a = Empty * Null
  If IsNull(a) Then Session.Echo "Null propagated through the expression."
End Sub
```

See Also Macro Control and Compilation on page 10

IsNumeric

Syntax **`IsNumeric(expression)`**

Description Returns **`True`** if **`expression`** can be converted to a number; returns **`False`** otherwise. If passed a number or a variant containing a number, then **`IsNumeric`** always returns **`True`**. If a string or string variant is passed, then IsNumeric will return True only if the string can be converted to a number. The following syntax is recognized as valid numbers:

```
&Hhexdigits[&|%|!|#|@]

&[O]octaldigits[&|%|!|#|@]
```

```
[-|+]digits[.[digits]][E[-|+]digits][!|%|&|#|@]
```

If an **Object** variant is passed, then the default property of that object is retrieved and one of the above rules is applied.

IsNumeric returns False if **expression** is a date.

Example

```
Sub Main
  Dim s$ As String
  s$ = InputBox("Enter a number.","Enter Number")
  If IsNumeric(s$) Then
    Session.Echo "You did well!"
  Else
    Session.Echo "You didn't do so well!"
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Numeric, Math, and Accounting Functions on page 9

IsObject

Syntax **IsObject(expression)**

Description Returns **True** if **expression** is a **Variant** variable containing an **Object**; returns **False** otherwise.

Example

```
Sub Main
  Dim v As Variant
  On Error Resume Next
  Set v = GetObject(,"Excel.Application")
  If IsObject(v) Then
    Session.Echo  "The default object value is: " & v = v.Value
  Else
    Session.Echo "Excel not loaded."
  End If
End Sub
```

See Also Objects on page 18

Item$

Syntax **Item$(text$,first [,[last] [,delimiters$]])**

Description Returns all the items between **first** and **last** within the specified formatted text list. The **Item$** function takes the following parameters:

Parameter	Description
text$	String containing the text from which a range of items is returned.
first	Integer containing the index of the first item to be returned. If **first** is greater than the number of items in **text$**, then a zero-length string is returned.
last	Integer containing the index of the last item to be returned. All of the items between **first** and **last** are returned. If **last** is greater than the number of items in **text$**, then all items from **first** to the end of text are returned. If **last** is missing, then only the item specified by **first** is returned.
delimiters$	String containing different item delimiters. By default, items are separated by commas and end-of-lines. This can be changed by specifying different delimiters in the **delimiters$** parameter.

The **Item$** function treats embedded null characters as regular characters.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  ilist$ = "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15"
  slist$ = "1/2/3/4/5/6/7/8/9/10/11/12/13/14/15"
  list1$ = Item$(ilist$,5,12)
  list2$ = Item$(slist$,2,9,"/")
  Session.Echo "The returned lists are: " & crlf & list1$ & crlf & list2$
End Sub
```

See Also Character and String Manipulation on page 3

ItemCount

Syntax **ItemCount(text$ [,delimiters$])**

Description Returns an **Integer** containing the number of items in the specified delimited text. Items are substrings of a delimited text string. Items, by default, are separated by commas and/or end-of-lines. This can be changed by specifying different delimiters in the **delimiters$** parameter. For example, to parse items using a backslash:

```
n = ItemCount(text$,"\")
```

The **ItemCount** function treats embedded null characters as regular characters.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  ilist$ = "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15"
  slist$ = "1/2/3/4/5/6/7/8/9/10/11/12/13/14/15/16/17/18/19"
  l1% = ItemCount(ilist$)
  l2% = ItemCount(slist$,"/")
  mesg = "The first lists contains: " & l1% & " items." & crlf
```

```
  mesg = mesg & "The second list contains: " & l2% & " items."
  Session.Echo mesg
End Sub
```

See Also Character and String Manipulation on page 3

K

Keywords (topic)

The following *keywords* are any word or symbol recognized as part of the macro language.

`Access`	`DefLng`	`Like`	`Random`	`Xor`
`Alias`	`DefObj`	`Line`	`Read`	
`And`	`DefSng`	`ListBox`	`ReDim`	
`Any`	`DefStr`	`Lock`	`Rem`	
`Append`	`DefVar`	`Long`	`Resume`	
`Application`	`Dialog`	`Loop`	`Return`	
`As`	`Dim`	`LSet`	`RSet`	
`Base`	`Do`	`Mid`	`Seek`	
`Begin`	`Double`	`MidB`	`Select`	
`Binary`	`DropListBox`	`Mod`	`Session`	
`Boolean`	`Else`	`Name`	`Set`	
`ByRef`	`ElseIf`	`New`	`Shared`	
`ByVal`	`End`	`Next`	`Single`	
`Call`	`Eqv`	`Not`	`Spc`	
`CancelButton`	`Error`	`Nothing`	`Static`	
`Case`	`Exit`	`Object`	`StdCall`	
`CDecl`	`Explicit`	`Off`	`Step`	
`CheckBox`	`For`	`OKButton`	`Stop`	
`Chr`	`Function`	`On`	`String`	
`ChrB`	`Get`	`Open`	`Sub`	
`ChrW`	`Global`	`Option`	`System`	
`Circuit`	`GoSub`	`Optional`	`Tab`	
`Close`	`Goto`	`OptionButton`	`Text`	
`ComboBox`	`GroupBox`	`OptionGroup`	`TextBox`	
`Compare`	`HelpButton`	`Or`	`Then`	
`Const`	`If`	`Output`	`Time`	
`CStrings`	`Imp`	`ParamArray`	`To`	
`Currency`	`Inline`	`Pascal`	`Transfer`	
`Date`	`Input`	`Picture`	`Type`	
`Declare`	`Input`	`PictureButton`	`Unlock`	
`Default`	`InputB`	`Preserve`	`Until`	
`DefBool`	`Integer`	`Print`	`Variant`	
`DefCur`	`Is`	`Private`	`Wend`	
`DefDate`	`Len`	`Public`	`While`	
`DefDbl`	`Let`	`PushButton`	`Width`	
`DefInt`	`Lib`	`Put`	`Write`	

Restrictions

All keywords are reserved in that you cannot create a variable, function, constant, or subroutine with the same name as a keyword. However, you are free to use all keywords as the names of structure members.

For all other keywords, the following restrictions apply:

- You can create a subroutine or function with the same name as a keyword.
- You can create a variable with the same name as a keyword as long as the variable is first explicitly declared with a **Dim**, **Private**, or **Public** statement.

Kill

Syntax **Kill pathname**

Description Deletes all files matching **pathname**. The **Kill** statement accepts the following named parameter:

Parameter	Description
pathname	Specifies the file to delete. If **filetype** is specified, then this parameter must specify a path. Otherwise, this parameter can include both a path and a file specification containing wildcards.

The **pathname** argument can include wildcards, such as * and ?. The * character matches any sequence of zero or more characters, whereas the ? character matches any single character. Multiple *'s and ?'s can appear within the expression to form complex searching patterns.

Example

```
Sub Main
   If Not FileExists("test1.dat") Then
      Open "test1.dat" For Output As #1
      Open "test2.dat" For Output As #2
      Close
   End If
   If FileExists ("test1.dat") Then
      Session.Echo "File test1.dat exists."
      Kill "test?.dat"
   End If
   If FileExists ("test1.dat") Then
      Session.Echo "File test1.dat still exists."
   Else
      Session.Echo "test?.dat successfully deleted."
   End If
End Sub
```

See Also Drive, Folder, and File Access on page 4

L

Lbound

Syntax `Lbound(ArrayVariable() [,dimension])`

Description Returns an **`Integer`** containing the lower bound of the specified dimension of the specified array variable. The **`dimension`** parameter is an integer specifying the desired dimension. If this parameter is not specified, then the lower bound of the first dimension is returned.

The **`Lbound`** function can be used to find the lower bound of a dimension of an array returned by an OLE Automation method or property:

```
Lbound(object.property [,dimension])
Lbound(object.method [,dimension])
```

Examples This example dimensions two arrays and displays their lower bounds.

```
Sub Main
  Dim a(5 To 12)
  Dim b(2 To 100, 9 To 20)
  lba = LBound(a)
  lbb = LBound(b,2)
  Session.Echo "The lower bound of a is: " & lba & _
    " The lower bound of b is: " & lbb
  'This example uses LBound and UBound to dimension a
  'dynamic array to hold a copy of an array redimmed by the
  'FileList statement.
  Dim fl$()
  FileList fl$,"*.*"
  count = UBound(fl$)
  If ArrayDims(a) Then
    Redim nl$(LBound(fl$) To UBound(fl$))
    For x = 1 To count
      nl$(x) = fl$(x)
    Next x
    Session.Echo "The last element of the new array is: " & _
      nl$(count)
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

LCase, LCase$

Syntax **`LCase[$](string)`**

Description Returns the lowercase equivalent of the specified string. **`LCase$`** returns a **`String`**, whereas **`LCase`** returns a **`String`** variant. **`Null`** is returned if **`string`** is **`Null`**.

Example

```
Sub Main
  lname$ = "WILLIAMS"
  fl$ = Left$(lname$,1)
  rest$ = Mid$(lname$,2,Len(lname$))
  lname$ = fl$ & LCase$(rest$)
  Session.Echo "The converted name is: " & lname$
End Sub
```

See Also Character and String Manipulation on page 3

Left, Left$, LeftB, LeftB$

Syntax

```
Left[$](string, length)
LeftB[$](string,length)
```

Description Returns the leftmost **`length`** characters (for **`Left`** and **`Left$`**) or bytes (for **`LeftB`** and **`LeftB$`**) from a given string.

`Left$ returns a String, whereas Left returns a String variant.`

The **`length`** parameter is an **`Integer`** value specifying the number of characters to return. If **`length`** is 0, then a zero-length string is returned. If **`length`** is greater than or equal to the number of characters in the specified string, then the entire string is returned.

The **`LeftB`** and **`LeftB$`** functions are used to return a sequence of bytes from a string containing byte data. In this case, **`length`** specifies the number of bytes to return. If **`length`** is greater than the number of bytes in **`string`**, then the entire string is returned.

`Null` is returned if **`string`** is **`Null`**.

Example

```
Sub Main
  lname$ = "WILLIAMS"
  fl$ = Left$(lname$,1)
  rest$ = Mid$(lname$,2,Len(lname$))
  lname$ = fl$ & LCase$(rest$)
  Session.Echo "The converted name is: " & lname$
End Sub
```

See Also Character and String Manipulation on page 3

Len, LenB

Syntax
```
Len(expression)
LenB(expression)
```

Description Returns the number of characters (for **`Len`**) or bytes (for **`LenB`**) in **`String`** expression or the number of bytes required to store the specified variable. If **`expression`** evaluates to a string, then **`Len`** returns the number of characters in a given string or 0 if the string is empty. When used with a **`Variant`** variable, the length of the variant when converted to a **`String`** is returned. If **`expression`** is a **`Null`**, then **`Len`** returns a **`Null`** variant.

The **`LenB`** function is used to return the number of bytes in a given string. On SBCS systems, the **`LenB`** and **`Len`** functions are identical.

If used with a non-**`String`** or non-**`Variant`** variable, these functions return the number of bytes occupied by that data element.

When used with user-defined data types, these functions return the combined size of each member within the structure. Since variable-length strings are stored elsewhere, the size of each variable-length string within a structure is 2 bytes.

The following table describes the sizes of the individual data elements when appearing within a structure:

Data Element	Size
Integer	2 bytes
Long	4 bytes
Float	4 bytes
Double	8 bytes
Currency	8 bytes
String (variable-length)	2 bytes
String (fixed-length)	The length of the string as it appears in the string's declaration in characters for Len and bytes for LenB.
Objects	0 bytes. Both data object variables and variables of type object are always returned as 0 size.
User-defined type	Combined size of each structure member. Variable-length strings within structures require 2 bytes of storage. Arrays within structures are fixed in their dimensions. The elements for fixed arrays are stored within the structure and therefore require the number of bytes for each array element multiplied by the size of each array dimension: **`element_size*dimension1*dimension2...`**

`The Len` and **`LenB`** functions always returns 0 with object variables or any data object variable.

Examples This example uses the Len function to change uppercase names to lowercase with an uppercase first letter.

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  lname$ = "WILLIAMS"
  fl$ = Left$(lname$,1)
  ln% = Len(lname$)
  rest$ = Mid$(lname$,2,ln%)
  lname$ = fl$ & LCase$(rest$)
  Session.Echo "The converted name is: " & lname$

  'This example returns a table of lengths for standard numeric types.
  Dim lns(4)
  a% = 100 : b& = 200 : c! = 200.22 : d# = 300.22
  lns(1) = Len(a%)
  lns(2) = Len(b&)
  lns(3) = Len(c!)
  lns(4) = Len(d#)
  mesg = "Lengths of standard types:" & crlf
  mesg = mesg & "Integer: " & lns(1) & crlf
  mesg = mesg & "Long: " & lns(2) & crlf
  mesg = mesg & "Single: " & lns(3) & crlf
  mesg = mesg & "Double: " & lns(4) & crlf
  Session.Echo mesg
End Sub
```

See Also Character and String Manipulation on page 3

Let

Syntax `[Let] variable = expression`

Description Assigns the result of an expression to a variable. The use of the word `Let` is supported for compatibility with other implementations of VBA. Normally, this word is dropped.

When assigning expressions to variables, internal type conversions are performed automatically between any two numeric quantities. Thus, you can freely assign numeric quantities without regard to type conversions. However, it is possible for an overflow error to occur when converting from larger to smaller types. This happens when the larger type contains a numeric quantity that cannot be represented by the smaller type. For example, the following code will produce a runtime error:

```
Dim amount As Long
Dim quantity As Integer
amount = 400123    'Assign a value out of range for int.
quantity = amount    'Attempt to assign to Integer.
```

When performing an automatic data conversion, underflow is not an error.

Example

```
Sub Main
  Let a$ = "This is a string."
  Let b% = 100
  Let c# = 1213.3443
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Like

Syntax **`expression Like pattern`**

Description Compares two strings and returns **`True`** if the **`expression`** matches the given pattern; returns **`False`** otherwise. Case sensitivity is controlled by the **`Option Compare`** setting. The pattern expression can contain special characters that allow more flexible matching:

Character	Evaluates To
`?`	Matches a single character.
`*`	Matches one or more characters.
`#`	Matches any digit.
`[range]`	Matches if the character in question is within the specified range.
`[!range]`	Matches if the character in question is not within the specified range.

A **`range`** specifies a grouping of characters. To specify a match of any of a group of characters, use the syntax **`[ABCDE]`**. To specify a range of characters, use the syntax **`[A-Z]`**. Special characters must appear within brackets, such as **`[]*?#`**.

If **`expression`** or **`pattern`** is not a string, then both **`expression`** and **`pattern`** are converted to **`String`** variants and compared, returning a **`Boolean`** variant. If either variant is **`Null`**, then **`Null`** is returned.

The following table shows some examples:

Expression	True if pattern is	False if pattern is
`"EBW"`	`"E*W", "E*"`	`"E*B"`
`"SML"`	`"B*[r-t]icMacro"`	`"B[r-t]ic"`
`"Version"`	`"V[e]?s*n"`	`"V[r]?s*N"`
`"2.0"`	`"#.#","#?#"`	`"###","#?[!0-9]"`
`"[ABC]"`	`"[[]*]"`	`"[ABC]","[*]"`

Example

```
Sub Main
  a$ = "This is a string variable of 123456 characters"
  b$ = "123.45"
  If a$ Like "[A-Z][g-i]*" Then Session.Echo _
    "The first comparison is True."
  If b$ Like "##3.##" Then Session.Echo "_
```

```
    The second comparison is True."
  If a$ Like "*variable*" Then Session.Echo _
    "The third comparison is True."
End Sub
```

See Also Character and String Manipulation on page 3

Line Input#

Syntax `Line Input #filenumber,variable`

Description Reads an entire line into the given variable.

The **`filenumber`** parameter is a number that is used to refer to the open file the number passed to the **`Open`** statement. The **`filenumber`** must reference a file opened in **`Input`** mode.

The file is read up to the next end-of-line, but the end-of-line character(s) is (are) not returned in the string. The file pointer is positioned after the terminating end-of-line.

The **`variable`** parameter is any string or variant variable reference. This statement will automatically declare the variable if the specified variable has not yet been used or dimensioned.

This statement recognizes either a single line feed or a carriage-return/line-feed pair as the end-of-line delimiter.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Open "c:\autoexec.bat" For Input As #1
  For x = 1 To 5
    Line Input #1,lin$
    mesg = mesg & lin$ & crlf
  Next x
  Session.Echo "The first 5 lines of your autoexec.bat are:" & crlf & mesg
End Sub
```

See Also Drive, Folder, and File Access on page 4

Line Numbers (topic)

Line numbers are not supported. As an alternative to line numbers, you can use meaningful labels as targets for absolute jumps, as shown below:

```
Sub Main
  Dim i As Integer
  On Error Goto MyErrorTrap
  i = 0
LoopTop:
  i = i + 1
  If i < 10 Then Goto LoopTop
```

```
MyErrorTrap:
  Session.Echo "An error occurred."
End Sub
```

Line$

Syntax `Line$(text$,first[,last])`

Description Returns a **`String`** containing a single line or a group of lines between **`first`** and **`last`**. Lines are delimited by carriage return, line feed, or carriage-return/line-feed pairs. Embedded null characters are treated as regular characters. The **`Line$`** function takes the following parameters:

Parameter	Description
`text$`	String containing the text from which the lines will be extracted.
`first`	Integer representing the index of the first line to return. If **`last`** is omitted, then this line will be returned. If **`first`** is greater than the number of lines in **`text$`**, then a zero-length string is returned.
`last`	Integer representing the index of the last line to return.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Open "c:\autoexec.bat" For Input As #1
  For x = 1 To 5
    Line Input #1,lin$
    txt = txt & lin$ & crlf
  Next x
  lines$ = Line$(txt,3,4)
  Session.Echo lines$
End Sub
```

See Also Character and String Manipulation on page 3

LineCount

Syntax `LineCount(text$)`

Description Returns an **`Integer`** representing the number of lines in **`text$`**. Lines are delimited by carriage return, line feed, or both. Embedded null characters are treated as regular characters.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  x = 1
  Open "c:\autoexec.bat" For Input As #1
  While (x < 10) And Not EOF(1)
    Line Input #1,lin$
    txt = txt & lin$ & crlf
    x = x + 1
  Wend
```

```
  lines! = LineCount(txt)
  Session.Echo "The number of lines in txt is: " & lines! & crlf & crlf & txt
End Sub
```

See Also Character and String Manipulation on page 3

ListBox

Syntax `ListBox x,y,width,height,ArrayVariable,.Identifier`

Description Creates a listbox within a dialog template. When the dialog is invoked, the listbox will be filled with the elements contained in **`ArrayVariable`**. This statement can only appear within a dialog template (i.e., between the **`Begin Dialog`** and **`End Dialog`** statements). The **`ListBox`** statement requires the following parameters:

Parameter	Description
`x, y`	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
`width, height`	Integer coordinates specifying the dimensions of the control in dialog units.
`ArrayVariable`	Specifies a single-dimensioned array of strings used to initialize the elements of the listbox. If this array has no dimensions, then the listbox will be initialized with no elements. A runtime error results if the specified array contains more than one dimension. **`ArrayVariable`** can specify an array of any fundamental data type (structures are not allowed). null and empty values are treated as zero-length strings.
`.Identifier`	Name by which this control can be referenced by statements in a dialog function (such as **`DlgFocus`** and **`DlgEnable`**). This parameter also creates an integer variable whose value corresponds to the index of the listbox's selection (0 is the first item, 1 is the second, and so on), which is not affected by the current setting of the Option Base command. This variable can be accessed using the following syntax: **`DialogVariable.Identifier`**

Example

```
Sub Main
  Dim files() As String
  Dim dirs() As String
  Begin Dialog ListBoxTemplate 16,32,184,96,"Sample"
    Text 8,4,24,8,"&Files:"
    ListBox 8,16,60,72,files$,.Files
    Text 76,4,21,8,"&Dirs:"
    ListBox 76,16,56,72,dirs$,.Dirs
    OKButton 140,4,40,14
    CancelButton 140,24,40,14
  End Dialog
  FileList files
  FileDirs dirs
```

```
  Dim ListBoxDialog As ListBoxTemplate
  rc% = Dialog(ListBoxDialog)
End Sub
```

See Also User Interaction on page 16

Literals (topic)

Literals are values of a specific type. The following table shows the different types of literals:

Literal	Description
10	Integer whose value is 10.
43265	Long whose value is 43,265.
5#	Double whose value is 5.0. A number's type can be explicitly set using any of the following type-declaration characters: **%** Integer **&** long **#** double **!** single
5.5	Double whose value is 5.5. Any number with decimal point is considered a double.
5.4E100	Double expressed in scientific notation.
&HFF	Integer expressed in hexadecimal.
&O47	Integer expressed in octal.
&HFF#	Double expressed in hexadecimal.
"hello"	String of five characters: **hello**.
"""hello"""	String of seven characters: **"hello"**. Quotation marks can be embedded within strings by using two consecutive quotation marks.
#1/1/1994#	Date value whose internal representation is 34335.0. Any valid date can appear with **#**s. Date literals are interpreted at execution time using the locale settings of the host environment. To ensure that date literals are correctly interpreted for all locales, use the international date format: **YYYY-MM-DD HH:MM:SS#**

Constant folding

The compiler supports constant folding where constant expressions are calculated by the compiler at compile time. For example, the expression:

```
i% = 10 + 12
```

is the same as:

```
i% = 22
```

Similarly, with strings, the expression:

```
s$ = "Hello," + " there" + Chr(46)
```

is the same as:

```
s$ = "Hello, there."
```

Loc

Syntax `Loc(filenumber)`

Description Returns a `Long` representing the position of the file pointer in the given file. The `filenumber` parameter is an `Integer` used to refer to the number passed by the `Open` statement. The `Loc` function returns different values depending on the mode in which the file was opened:

File Mode	Returns
`Input`	Current byte position divided by 128
`Output`	Current byte position divided by 128
`Append`	Current byte position divided by 128
`Binary`	Position of the last byte read or written
`Random`	Number of the last record read or written

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Open "c:\autoexec.bat" For Input As #1
  For x = 1 To 5
    If Not EOF(1) Then Line Input #1,lin$
  Next x
  lc% = Loc(1)
  Close
  Session.Echo "The file location is: " & lc%
End Sub
```

See Also Drive, Folder, and File Access on page 4

Lock, Unlock

Syntax

```
Lock [#] filenumber [,{record | [start] To end}]
Unlock [#] filenumber [,{record | [start] To end}]
```

Description Locks or unlocks a section of the specified file, granting or denying other processes access to that section of the file. The `Lock` statement locks a section of the specified file, preventing other processes from accessing that section of the file until the `Unlock` statement is issued. The `Unlock` statement unlocks a section of the specified file, allowing other processes access to that section of the file. The `Lock` and `Unlock` statements require the following parameters:

Parameter	Description
filenumber	Integer used to refer to the open file—the number passed to the **Open** statement.
record	Long specifying which record to lock or unlock.
start	Long specifying the first record within a range to be locked or unlocked.
end	Long specifying the last record within a range to be locked or unlocked.

For sequential files, the **record**, **start**, and **end** parameters are ignored. The entire file is locked or unlocked.

The section of the file is specified using one of the following:

Syntax	Description
No parameters	Locks or unlocks the entire file (no record specification is given).
record	Locks or unlocks the specified record number (for Random files) or byte (for Binary files).
To end	Locks or unlocks from the beginning of the file to the specified record (for Random files) or byte (for Binary files).
start To end	Locks or unlocks the specified range of records (for Random files) or bytes (for Binary files).

The lock range must be the same as that used to subsequently unlock the file range, and all locked ranges must be unlocked before the file is closed. Ranges within files are not unlocked automatically when your macro terminates, which can cause file access problems for other processes. It is a good idea to group the **Lock** and **Unlock** statements close together in the code, both for readability and so subsequent readers can see that the lock and unlock are performed on the same range. This practice also reduces errors in file locks.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  a$ = "This is record number: "
  b$ = "0"
  rec$ = ""
  mesg = ""
  Open "test.dat" For Random Access Write Shared As #1
  For x = 1 To 10
    rec$ = a$ & x
    Lock #1,x
    Put #1,,rec$
    Unlock #1,x
    mesg = mesg & rec$ & crlf
  Next x
  Close
  Session.Echo "The records are:" & crlf & mesg
  mesg = ""
  Open "test.dat" For Random Access Read Write Shared As #1
  For x = 1 To 10
```

```
    rec$ = Mid$(rec$,1,23) & (11 - x)
    Lock #1,x
    Put #1,x,rec$
    Unlock #1,x
    mesg = mesg & rec$ & crlf
  Next x
  Session.Echo "The records are: " & crlf & mesg
  Close
  Kill "test.dat"
End Sub
```

See Also Drive, Folder, and File Access on page 4

Lof

Syntax `Lof(filenumber)`

Description Returns a **`Long`** representing the number of bytes in the given file. The **`filenumber`** parameter is an **`Integer`** used to refer to the open file the number passed to the **`Open`** statement. The file must currently be open.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  a$ = "This is record number: "
  Open "test.dat" For Random Access Write Shared As #1
  For x = 1 To 10
    rec$ = a$ & x
    put #1,,rec$
    mesg = mesg & rec$ & crlf
  Next x
  Close
  Open "test.dat" For Random Access Read Write Shared As #1
  r% = Lof(1)
  Close
  Session.Echo "The length of test.dat is: " & r%
End Sub
```

See Also Drive, Folder, and File Access on page 4

Log

Syntax `Log(number)`

Description Returns a **`Double`** representing the natural logarithm of a given number. The value of **`number`** must be a **`Double`** greater than 0. The value of **`e`** is 2.71828.

Example

```
Sub Main
  x# = Log(100)
  Session.Echo "The natural logarithm of 100 is: " & x#
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

Long (data type)

Syntax **`Long`**

Description **`Long`** variables are used to hold numbers (with up to ten digits of precision) within the following range:

```
–2,147,483,648 <= Long <= 2,147,483,647
```

Internally, longs are 4-byte values. Thus, when appearing within a structure, longs require 4 bytes of storage. When used with binary or random files, 4 bytes of storage are required.

The type-declaration character for **`Long`** is &.

See Also Keywords, Data Types, Operators, and Expressions on page 6

LSet

Syntax 1 **`LSet dest = source`**

Syntax 2 **`LSet dest_variable = source_variable`**

Description Left-aligns the source string in the destination string or copies one user-defined type to another.

Syntax 1

The **`LSet`** statement copies the source string **`source`** into the destination string **`dest`**. The **`dest`** parameter must be the name of either a **`String`** or **`Variant`** variable. The **`source`** parameter is any expression convertible to a string.

If **`source`** is shorter in length than **`dest`**, then the string is left-aligned within **`dest`**, and the remaining characters are padded with spaces. If **`source$`** is longer in length than **`dest`**, then **`source`** is truncated, copying only the leftmost number of characters that will fit in **`dest`**.

The **`destvariable`** parameter specifies a **`String`** or **`Variant`** variable. If **`destvariable`** is a **`Variant`** containing **`Empty`**, then no characters are copied. If **`destvariable`** is not convertible to a **`String`**, then a runtime error occurs. A runtime error results if **`destvariable`** is **`Null`**.

Syntax 2

The source structure is copied byte for byte into the destination structure. This is useful for copying structures of different types. Only the number of bytes of the smaller of the two structures is copied. Neither the source structure nor the destination structure can contain strings.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Dim mesg, tmpstr$
  tmpstr$ = String$(40, "*")
  mesg = "Here are two strings that have been right-" + crlf
  mesg = mesg & "and left-justified in a 40-character string."
```

```
  mesg = mesg & crlf & crlf
  RSet tmpstr$ = "Right->"
  mesg = mesg & tmpstr$ & crlf
  LSet tmpstr$ = "<-Left"
  mesg = mesg & tmpstr$ & crlf
  Session.Echo mesg
End Sub
```

See Also Character and String Manipulation on page 3

LTrim, LTrim$

See Trim, Trim$, LTrim, LTrim$, RTrim, RTrim$.

M

Mid, Mid$, MidB, MidB$ (functions)

Syntax

```
Mid[$](string, start [,length])
MidB[$](string, start [,length])
```

Description Returns a substring of the specified string, beginning with **`start`**, for **`length`** characters (for **`Mid`** and **`Mid$`**) or bytes (for **`MidB`** and **`MidB$`**).

The **`Mid`** and **`Mid$`** functions return a substring starting at character position **`start`** and will be **`length`** characters long. The **`MidB`** and **`MidB`** functions return a substring starting at byte position **`start`** and will be **`length`** bytes long.

The **`Mid$`** and **`MidB$`** functions return a string, whereas the **`Mid`** and **`MidB`** functions return a string variant.

These functions take the following named parameters:

Parameter	Description
`string`	Any string expression containing the text from which data is returned.
`start`	Integer specifying the position where the substring begins. If **`start`** is greater than the length of **`string`**, then a zero-length string is returned.
`length`	Integer specifying the number of characters or bytes to return. If this parameter is omitted, then the entire string is returned, starting at **`start`**.

The **`Mid`** function will return **`Null`** if **`string`** is **`Null`**.

The **`MidB`** and **`MidB$`** functions are used to return a substring of bytes from a string containing byte data.

Example

```
Const crlf = Chr$(13) + Chr$(10)
```

```
Sub Main
  a$ = "This is the Main string containing text."
  b$ = Mid$(a$,13,Len(a$))
  Mid$ (b$,1) = NEW "
  Session.Echo a$ & crlf & b$
End Sub
```

See Also Character and String Manipulation on page 3

Mid, Mid$, MidB, MidB$ (statements)

Syntax

```
Mid[$](variable,start[,length]) = newvalue
MidB[$](variable,start[,length]) = newvalue
```

Description Replaces one part of a string with another. The **Mid**/**Mid$** statements take the following parameters:

Parameter	Description
variable	String or variant variable to be changed.
start	Integer specifying the character position (for Mid and Mid$) or byte position (for MidB and MidB$) within **variable** where replacement begins. If **start** is greater than the length of **variable**, then **variable** remains unchanged.
length	Integer specifying the number of characters or bytes to change. If this parameter is omitted, then the entire string is changed, starting at **start**.
newvalue	Expression used as the replacement. This expression must be convertible to a string.

The resultant string is never longer than the original length of **variable**.

With **Mid** and **MidB**, **variable** must be a variant variable convertible to a string, and **newvalue** is any expression convertible to a string. A runtime error is generated if either variant is null.

The **MidB** and **MidB$** statements are used to replace a substring of bytes, whereas **Mid** and **Mid$** are used to replace a substring of characters.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  a$ = "This is the Main string containing text."
  b$ = Mid$(a$,13,Len(a$))
  Mid$(b$,1) = "NEW "
  Session.Echo a$ & crlf & b$
End Sub
```

See Also Character and String Manipulation on page 3

Minute

Syntax `Minute(time)`

Description Returns the minute of the day encoded in the specified **time** parameter. The value returned is as an **Integer** between 0 and 59 inclusive. The **time** parameter is any expression that converts to a date.

Example

```
Sub Main
  xt# = TimeValue(Time$())
  xh# = Hour(xt#)
  xm# = Minute(xt#)
  xs# = Second(xt#)
  Session.Echo "The current time is: " & xh# & ":" & xm# & ":" & xs#
End Sub
```

See Also Time and Date Access on page 17

MIRR

Syntax `MIRR(valuearray(),financerate,reinvestrate)`

Description Returns a **Double** representing the modified internal rate of return for a series of periodic payments and receipts. The modified internal rate of return is the equivalent rate of return on an investment in which payments and receipts are financed at different rates. The interest cost of investment and the rate of interest received on the returns on investment are both factors in the calculations. The **MIRR** function requires the following named parameters:

Parameter	Description
valuearray()	Array of double numbers representing the payments and receipts. Positive values are payments (invested capital), and negative values are receipts (returns on investment). There must be at least one positive (investment) value and one negative (return) value.
financerate	Double representing the interest rate paid on invested monies (paid out).
reinvestrate	Double representing the rate of interest received on incomes from the investment (receipts).

The **financerate** and **reinvestrate** parameters should be expressed as percentages. For example, 11 percent should be expressed as 0.11.

To return the correct value, be sure to order your payments and receipts in the correct sequence.

Example This example illustrates the purchase of a lemonade stand for $800 financed with money borrowed at 10%. The returns are estimated to accelerate as the stand gains popularity. The proceeds are placed in a bank at 9 percent interest. The incomes are estimated (generated) over 12 months. This program first generates the income stream array in two **For...Next** loops, and then the modified internal rate of return is calculated and displayed. Notice that the annual rates are normalized to monthly rates by dividing them by 12.

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Dim valu#(12)
  valu(1) = -800          'Initial investment
  mesg = valu(1) & ", "
  For x = 2 To 5
    valu(x) = 100 + (x * 2)    'Incomes months 2-5
    mesg = mesg & valu(x) & ", "
  Next x
  For x = 6 To 12
    valu(x) = 100 + (x * 10)    'Incomes months 6-12
    mesg = mesg & valu(x) & ", "
  Next x
  retrn# = MIRR(valu,.1/12,.09/12)  'Note: normalized annual rates
  mesg = "The values: " & crlf & mesg & crlf & crlf
  Session.Echo mesg & "Modified rate: " & Format(retrn#,"Percent")
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

MkDir

Syntax **`MkDir path`**

Description Creates a new directory as specified by **`path`**.

Example

```
Sub Main
  On Error Resume Next
  MkDir "TestDir"
  If Err <> 0 Then
    Session.Echo "The following error occurred: " & Error(Err)
  Else
    Session.Echo "Directory was created and is about to be removed."
    RmDir "TestDir"
  End If
End Sub
```

See Also Drive, Folder, and File Access on page 4

Mod

Syntax **`expression1 Mod expression2`**

Description Returns the remainder of **`expression1`** / **`expression2`** as a whole number. If both expressions are integers, then the result is an integer. Otherwise, each expression is converted to a **`Long`** before performing the operation, returning a **`Long`**. A runtime error occurs if the result overflows the range of a long. If either expression is null, then null is returned. Empty is treated as 0.

Example This example uses the Mod operator to determine the value of a randomly selected card where card 1 is the ace (1) of clubs and card 52 is the king (13) of spades. Since the values recur in a sequence of 13 cards within 4 suits, we can use the **`Mod`** function to determine the value of any given card number.

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  cval$ = "ACE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,"
  cval$ = cval$+"NINE,TEN,JACK,QUEEN,KING"
  Randomize
  card% = Random(1,52)
  value = card% Mod 13
  If value = 0 Then value = 13
  CardNum$ = Item$(cval,value)
  If card% < 53 Then suit$ = "spades"
  If card% < 40 Then suit$ = "hearts"
  If card% < 27 Then suit$ = "diamonds"
  If card% < 14 Then suit$ = "clubs"
  mesg = "Card number " & card% & " is the "
  mesg = mesg & CardNum & " of " & suit$
  Session.Echo mesg
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Numeric, Math, and Accounting Functions on page 9

Month

Syntax `Month(date)`

Description Returns the month of the date encoded in the specified **`date`** parameter. The value returned is as an **`Integer`** between 1 and 12 inclusive. The **`date`** parameter is any expression that converts to a date.

Example

```
Sub Main
  mons$ = "Jan., Feb., Mar., Apr., May, Jun., Jul., "
  mons$ = mons$ + "Aug., Sep., Oct., Nov., Dec."
  tdate$ = Date$
  tmonth! = Month(DateValue(tdate$))
  Session.Echo "The current month is: " & Item$(mons$,tmonth!)
End Sub
```

See Also Time and Date Access on page 17

Msg (object)

The Msg object provides a quick modeless dialog—that is, a dialog which the user may ignore, continuing to run other commands before closing. A good example of a modeless dialog is the Edit>Find dialog in many word processors, which can be left open while editing the text.

Msg.Close

Syntax `Msg.Close`

Description Closes the modeless message dialog. Nothing will happen if there is no open message dialog.

Example

```
Sub Main
  Msg.Open "Printing. Please wait...",0,True,True
  Sleep 3000
  Msg.Close
End Sub
```

See Also User Interaction on page 16

Msg.Open

Syntax **`Msg.Open prompt,timeout,cancel,thermometer [,XPos,YPos]`**

Description Displays a message in a dialog with an optional Cancel button and thermometer. The **`Msg.Open`** method takes the following named parameters:

Parameter	Description
`prompt`	String containing the text to be displayed. The text can be changed using the Msg.Text property.
`timeout`	Integer specifying the number of seconds before the dialog is automatically removed. The **`timeout`** parameter has no effect if its value is 0.
`cancel`	Boolean controlling whether or not a Cancel button appears within the dialog beneath the displayed message. If this parameter is True, then a Cancel button appears. If it is not specified or False, then no Cancel button is created. If a user chooses the Cancel button at runtime, a trappable runtime error is generated (error number 18). In this manner, a message dialog can be displayed and processing can continue as normal, aborting only when the user cancels the process by choosing the Cancel button.
`thermometer`	Boolean controlling whether the dialog contains a thermometer. If this parameter is True, then a thermometer is created between the text and the optional Cancel button. The thermometer initially indicates 0% complete and can be changed using the Msg.Thermometer property.
`XPos`, **`YPos`**	Integer coordinates specifying the location of the upper left corner of the message box, in twips (twentieths of a point). If these parameters are not specified, then the window is centered on top of the application.

Unlike other dialoges, a message dialog remains open until the user selects Cancel, the timeout has expired, or the **`Msg.Close`** method is executed (this is sometimes referred to as modeless).

Only a single message window can be opened at any one time. The message window is removed automatically when a macro terminates.

The Cancel button, if present, can be selected using either the mouse or keyboard. However, these events will never reach the message dialog unless you periodically call DoEvents from within your macro.

Example

```
Sub Main
  Msg.Open "Printing. Please wait...",0,True,False
  Sleep 3000
  Msg.Close
  Msg.Open "Printing. Please wait...",0,True,True
  For x = 1 to 100
    Msg.Thermometer = x
  Next x
  Sleep 1000
  Msg.Close
End Sub
```

See Also User Interaction on page 16

Msg.Text

Syntax **`Msg.Text [= newtext$]`**

Description Changes the text within an open message dialog (one that was previously opened with the **`Msg.Open`** method). The message dialog is not resized to accommodate the new text. A runtime error will result if a message dialog is not currently open (using Msg.Open).

Example

```
Sub Main
  Msg.Open "Reading Record",0,True,False
  For i = 1 To 100
    'Read a record here.
    'Update the modeless message box.
    Sleep 100
    Msg.Text ="Reading record " & i
  Next i
  Msg.Close
End Sub
```

See Also User Interaction on page 16

Msg.Thermometer

Syntax **`Msg.Thermometer [= percentage]`**

Description Changes the percentage filled indicated within the thermometer of a message dialog (one that was previously opened with the **`Msg.Open`** method). A runtime error will result if a message box is not currently open (using **`Msg.Open`**) or if the value of **`percentage`** is not between 0 and 100 inclusive.

Example

```
Sub Main
  On Error Goto ErrorTrap
  Msg.Open "Reading records from file...",0,True,True
  For i = 1 To 100    'Read a record here.
            'Update the modeless message box.
    Msg.Thermometer =i
    DoEvents
    Sleep 50
  Next i
  Msg.Close
  On Error Goto 0    'Turn error trap off.
  Exit Sub
```

```
ErrorTrap:
  If Err = 809 Then
    MsgBox "Cancel was pressed!"
    Exit Sub      'Reset error handler.
  End If
End Sub
```

See Also User Interaction on page 16

MsgBox (function)

Syntax **`MsgBox(prompt [, [buttons] [,[title] [,helpfile,context]]])`**

Description Displays a message in a dialog with a set of predefined buttons, returning an **`Integer`** representing which button was selected. The **`MsgBox`** function takes the following named parameters:

Parameter	Description
`prompt`	Message to be displayed—any expression convertible to a string. End-of-lines can be used to separate lines (either a carriage return, line feed, or both). If a given line is too long, it will be word-wrapped. If **`prompt`** contains character 0, then only the characters up to the character 0 will be displayed. The width and height of the dialog are sized to hold the entire contents of **`prompt`**. A runtime error is generated if **`prompt`** is null.
`buttons`	Integer specifying the type of dialog (see below).
`title`	Caption of the dialog. This parameter is any expression convertible to a string. If it is omitted, then "SmarTerm" is used. A runtime error is generated if **`title`** is null.
`helpfile`	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then **`context`** must also be specified.
`context`	Number specifying the ID of the topic within **`helpfile`** for this dialog's help. If this parameter is specified, then **`helpfile`** must also be specified.

The **`MsgBox`** function returns one of the following values:

Constant	Value	Description
`ebOK`	1	OK was pressed.
`ebCancel`	2	Cancel was pressed.
`ebAbort`	3	Abort was pressed.
`ebRetry`	4	Retry was pressed.
`ebIgnore`	5	Ignore was pressed.
`ebYes`	6	Yes was pressed.
`ebNo`	7	No was pressed.

The **`buttons`** parameter is the sum of any of the following values:

Constant	Value	Description
`ebOKOnly`	0	Displays OK button only.
`ebOKCancel`	1	Displays OK and Cancel buttons.
`ebAbortRetryIgnore`	2	Displays Abort, Retry, and Ignore buttons.
`ebYesNoCancel`	3	Displays Yes, No, and Cancel buttons.
`ebYesNo`	4	Displays Yes and No buttons.
`ebRetryCancel`	5	Displays Retry and Cancel buttons.
`ebCritical`	16	Displays stop icon.
`ebQuestion`	32	Displays question mark icon.
`ebExclamation`	48	Displays exclamation point icon.
`ebInformation`	64	Displays information icon.
`ebDefaultButton1`	0	First button is the default button.
`ebDefaultButton2`	256	Second button is the default button.
`ebDefaultButton3`	512	Third button is the default button.
`ebApplicationModal`	0	Application modal—the current application is suspended until the dialog is closed.
`ebSystemModal`	4096	System modal—all applications are suspended until the dialog is closed.

The default value for **`buttons`** is 0 (display only the OK button, making it the default).

If both the **`helpfile`** and **`context`** parameters are specified, then context-sensitive help can be invoked using the help key F1. Invoking help does not remove the dialog.

Breaking Text across Lines

The **`prompt`** parameter can contain end-of-line characters, forcing the text that follows to start on a new line. The following example shows how to display a string on two lines:

```
MsgBox "This is on" + Chr(13) + Chr(10) + "two lines."
```

The carriage-return or line-feed characters can be used by themselves to designate an end-of-line.

Example

```
Sub Main
  MsgBox "This is a simple message box."
  MsgBox "This is a message box with a title and an icon.", _
    ebExclamation,"Simple"
  MsgBox "This message box has OK and Cancel buttons.", _
    ebOkCancel,"MsgBox"
  MsgBox "This message box has Abort, Retry, and Ignore buttons.", _
    ebAbortRetryIgnore,"MsgBox"
  MsgBox "This message box has Yes, No, and Cancel buttons.", _
    ebYesNoCancel Or ebDefaultButton2,"MsgBox"
  MsgBox "This message box has Yes and No buttons.",ebYesNo,"MsgBox"
```

```
  MsgBox "This message box has Retry and Cancel buttons." , _
    ebRetryCancel,"MsgBox"
  MsgBox "This message box is system modal!",ebSystemModal
End Sub
```

See Also User Interaction on page 16

MsgBox (statement)

Syntax `MsgBox prompt [, [buttons] [,[title] [, helpfile, context]]]`

Description Same as the **`MsgBox`** function, except that the statement form does not return a value. See **`MsgBox`** (function).

Example

```
Sub Main
  MsgBox "This is text displayed in a message box."  'Display text.
  MsgBox "The result is: " & (10 * 45)      'Display a number.
End Sub
```

See Also User Interaction on page 16

N

Name

Syntax `Name oldfile$ As newfile$`

Description Renames a file. Each parameter must specify a single filename. Wildcard characters such as * and ? are not allowed. You can name files to different directories on the same physical disk volume. For example, the following rename will work under Windows:

```
Name "c:\samples\mydoc.txt" As "c:\backup\doc\mydoc.bak"
```

You cannot rename files across physical disk volumes. For example, the following will error under Windows:

```
Name "c:\samples\mydoc.txt" As "a:\mydoc.bak"
```

To rename a file to a different physical disk, you must first copy the file, then erase the original:

```
FileCopy "c:\samples\mydoc.txt","a:\mydoc.bak"
Kill "c:\samples\mydoc.txt"
```

Example

```
Sub Main
  On Error Resume Next
  If FileExists("test.dat") Then
    Name "test.dat" As "test2.dat"
    If Err <> 0 Then
      mesg = "File exists and cannot be renamed! Error: " _
        & Err
    Else
      mesg = "File exists and renamed to test2.dat."
    End If
  Else
    Open "test.dat" For Output As #1
    Close
    Name "test.dat" As "test2.dat"
    If Err <> 0 Then
      mesg = "File created but not renamed! Error: " & Err
    Else
      mesg = "File created and renamed to test2.dat."
```

```
    End If
  End If
  Session.Echo mesg
End Sub
```

See Also Drive, Folder, and File Access on page 4

Named Parameters (topic)

Many language elements support named parameters. Named parameters allow you to specify parameters to a function or subroutine by name rather than in adherence to a predetermined order. The following table contains examples showing various calls to **`Session.Echo`** both using parameter by both name and position.

Parameter	Call
By Name	`DateAdd(Interval:= "m", Number:= 2, Date:= "December 31, 1992")`
By Position	`DateAdd("m", 2, "December 31, 1992")`

Using named parameter makes your code easier to read, while at the same time removes you from knowing the order of parameter. With functions that require many parameters, most of which are optional, code becomes significantly easier to write and maintain.

When supported, the names of the named parameter appear in the description of that language element.

When using named parameter, you must observe the following rules:

- Named parameter must use the parameter name as specified in the description of that language element. Unrecognized parameter names cause compiler errors.
- All parameters, whether named or positional, are separated by commas.
- The parameter name and its associated value are separated with **`:=`**
- If one parameter is named, then all subsequent parameters must also be named as shown here:

```
DateAdd("m", Number:= 2, Date:= "December 31, 1992")
DateAdd(Interval:= "m",,"December 31, 1992")    WRONG!!!
```

New

Syntax 1 **`Dim ObjectVariable As New ObjectType`**

Syntax 2 **`Set ObjectVariable = New ObjectType`**

Description Creates a new instance of the specified object type, assigning it to the specified object variable. The **`New`** keyword is used to declare a new instance of the specified data object. This keyword can only be used with data object types. At runtime, the application or extension that defines that object type is notified that a new object is being defined. The application responds by creating a new physical object (within the appropriate context) and returning a reference to that object, which is immediately assigned to the variable being declared. When that variable goes out of scope (i.e., the **Sub** or **Function** procedure in which the variable is declared ends), the application is notified. The application then performs some appropriate action, such as destroying the physical object.

See Also Objects on page 18

Not

Syntax **Not expression**

Description Returns either a logical or binary negation of **expression**. The result is determined as shown in the following table:

Expression	Result
True	False
False	True
Null	Null
Any numeric type	Binary negation of the number. If the number is an integer, then an integer is returned. Otherwise, the expression is first converted to a long, then a binary negation is performed, returning a long.
Empty	Treated as a long value 0.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  a = False
  b = True
  If (Not a and b) Then mesg = "a = False, b = True" & crlf
  toggle% = True
  mesg = mesg & "toggle% is now " & Format(toggle%,"True/False") & crlf
  toggle% = Not toggle%
  mesg = mesg & "toggle% is now " & Format(toggle%,"True/False") & crlf
  toggle% = Not toggle%
  mesg = mesg & "toggle% is now " & Format(toggle%,"True/False")
  Session.Echo mesg
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Now

Syntax **Now[()]**

Description Returns a **Date** variant representing the current date and time.

Example

```
Sub Main
  t1# = Now()
  Session.Echo "Wait a while and click OK."
  t2# = Now()
  t3# = Second(t2#) - Second(t1#)
  Session.Echo "Elapsed time was: " & t3# & " seconds."
End Sub
```

See Also Time and Date Access on page 17

NPer

Syntax `NPer(rate, pmt, pv, fv, due)`

Description Returns the number of periods for an annuity based on periodic fixed payments and a constant rate of interest. An annuity is a series of fixed payments paid to or received from an investment over a period of time. Examples of annuities are mortgages, retirement plans, monthly savings plans, and term loans. The **NPer** function requires the following named parameters:

Parameter	Description
rate	Double representing the interest rate per period. If the periods are monthly, be sure to normalize annual rates by dividing them by 12.
Pmt	Double representing the amount of each payment or income. Income is represented by positive values, whereas payments are represented by negative values.
Pv	Double representing the present value of your annuity. In the case of a loan, the present value would be the amount of the loan, and the future value (see below) would be zero.
Fv	Double representing the future value of your annuity. In the case of a loan, the future value would be zero, and the present value would be the amount of the loan.
Due	Integer indicating when payments are due for each payment period. A 0 specifies payment at the end of each period, whereas a 1 indicates payment at the start of each period.

Positive numbers represent cash received, whereas negative numbers represent cash paid out.

Example This example calculates the number of $100.00 monthly payments necessary to accumulate $10,000.00 at an annual rate of 10%. Payments are made at the beginning of the month.

```
Sub Main
  ag# = NPer((.10/12),100,0,10000,1)
  Session.Echo "The number of monthly periods is: " & Format(ag#,"Standard")
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

Npv

Syntax `Npv(rate, valuearray())`

Description Returns the net present value of an annuity based on periodic payments and receipts, and a discount rate. The **`Npv`** function requires the following named parameters:

Parameter	Description
`rate`	Double that represents the interest rate over the length of the period. If the values are monthly, annual rates must be divided by 12 to normalize them to monthly rates.
`valuearray()`	Array of double numbers representing the payments and receipts. Positive values are payments, and negative values are receipts. There must be at least one positive and one negative value.

Positive numbers represent cash received, whereas negative numbers represent cash paid out.

For accurate results, be sure to enter your payments and receipts in the correct order because **`Npv`** uses the order of the array values to interpret the order of the payments and receipts.

If your first cash flow occurs at the beginning of the first period, that value must be added to the return value of the **`Npv`** function. It should not be included in the array of cash flows.

`Npv` differs from the **`Pv`** function in that the payments are due at the end of the period and the cash flows are variable. **`Pv`**'s cash flows are constant, and payment may be made at either the beginning or end of the period.

Example This example illustrates the purchase of a lemonade stand for $800 financed with money borrowed at 10%. The returns are estimated to accelerate as the stand gains popularity. The incomes are estimated (generated) over 12 months. This program first generates the income stream array in two **`For...Next`** loops, and then the net present value (**`Npv`**) is calculated and displayed. Note normalization of the annual 10% rate.

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Dim valu#(12)
  valu(1) = -800        'Initial investment
  mesg = valu(1) & ", "
  For x = 2 To 5        'Months 2-5
    valu(x) = 100 + (x * 2)
    mesg = mesg & valu(x) & ", "
  Next x
  For x = 6 To 12      'Months 6-12
    valu(x) = 100 + (x * 10)  'Accelerated income
    mesg = mesg & valu(x) & ", "
  Next x
  NetVal# = NPV((.10/12),valu)
```

```
  mesg = "The values:" & crlf & mesg & crlf & crlf
  Session.Echo mesg & "Net present value: " & Format(NetVal#,"Currency")
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

O

Object (data type)

Syntax `Object`

Description Used to declare OLE Automation variables. The **`Object`** type is used to declare variables that reference objects within an application using OLE Automation. Each object is a 4-byte (32-bit) value that references the object internally. The value 0 (or **`Nothing`**) indicates that the variable does not reference a valid object, as is the case when the object has not yet been given a value. Accessing properties or methods of such **`Object`** variables generates a runtime error.

Using objects

`Object` variables are declared using the **`Dim`**, **`Public`**, or **`Private`** statement:

```
Dim MyApp As Object
```

`Object` variables can be assigned values (thereby referencing a real physical object) using the **`Set`** statement:

```
Set MyApp = CreateObject("phantom.application")
Set MyApp = Nothing
```

Properties of an **`Object`** are accessed using the dot (.) separator:

```
MyApp.Color = 10
i% = MyApp.Color
```

Methods of an **`Object`** are also accessed using the dot (.) separator:

```
MyApp.Open "sample.txt"
isSuccess = MyApp.Save("new.txt",15)
```

Automatic destruction

The compiler keeps track of the number of variables that reference a given object so that the object can be destroyed when there are no longer any references to it:

```
Sub Main()          'Number of references to object
  Dim a As Object              '0
  Dim b As Object              '0
  Set a = CreateObject("phantom.application)    '1
  Set b = a                  '2
  Set a = Nothing              '1
End Sub                  '0bject destroyed
```

Note An OLE Automation object is instructed by the compiler to destroy itself when no variables reference that object. However, it is the responsibility of the OLE Automation server to destroy it. Some servers do not destroy their objects, usually when the objects have a visual component and can be destroyed manually by the user.

See Also Objects on page 18

Objects (topic)

The macro language defines two types of objects: data objects and OLE Automation objects. Syntactically, these are referenced in the same way.

What is an object

An object is an encapsulation of data and routines into a single unit. The use of objects has the effect of grouping together a set of functions and data items that apply only to a specific object type.

Objects expose data items for programmability called properties. For example, a sheet object may expose an integer called **`NumColumns`**. Usually, properties can be both retrieved (get) and modified (set).

Objects also expose internal routines for programmability called methods. An object method can take the form of a function or a subroutine. For example, a OLE Automation object called **`MyApp`** may contain a method subroutine called **`Open`** that takes a single argument (a filename): **`MyApp.Open "c:\files\sample.txt"`**.

Declaring Object Variables

In order to gain access to an object, you must first declare an object variable using either **`Dim`**, **`Public`**, or **`Private`**: **`Dim o As Object`**. Initially, objects are given the value **`0`** (or **`Nothing`**). Before an object can be accessed, it must be associated with a existing object.

Assigning a Value to an Object Variable

An object variable must reference a real physical object before accessing any properties or methods of that object. To instantiate an object, use the `Set` statement.

```
Dim MyApp As Object
Set MyApp = CreateObject("Server.Application")
```

Accessing Object Properties

Once an object variable has been declared and associated with a physical object, it can be modified using macro code. Properties are syntactically accessible using the dot operator, which separates an object name from the property being accessed:

```
MyApp.BackgroundColor = 10
i% = MyApp.DocumentCount
```

Properties are set using the normal assignment statement:

```
MyApp.BackgroundColor = 10
```

Object properties can be retrieved and used within expressions:

```
i% = MyApp.DocumentCount + 10
Session.Echo "Number of documents = " & MyApp.DocumentCount
```

Accessing Object Methods

Like properties, methods are accessed via the dot operator. Object methods that do not return values behave like subroutines (i.e., the arguments are not enclosed within parentheses):

```
MyApp.Open "c:\files\sample.txt",True,15
```

Object methods that return a value behave like function calls. Any arguments must be enclosed in parentheses:

```
If MyApp.DocumentCount = 0 Then Session.Echo "No open documents."
NumDocs = app.count(4,5)
```

There is no syntactic difference between calling a method function and retrieving a property value, as shown below:

```
variable = object.property(arg1,arg2)
variable = object.method(arg1,arg2)
```

Comparing Object Variables

The values used to represent objects are meaningless to the macro in which they are used, with the following exceptions:

- Objects can be compared to each other to determine whether they refer to the same object.
- Objects can be compared with **`Nothing`** to determine whether the object variable refers to a valid object.

Object comparisons are accomplished using the **`Is`** operator:

```
If a Is b Then Session.Echo "a and b are the same object."
If a Is Nothing Then Session.Echo "a is not initialized."
If b Is Not Nothing Then Session.Echo "b is in use."
```

Collections

A collection is a set of related object variables. Each element in the set is called a member and is accessed via an index, either numeric or text, as shown below:

```
MyApp.Toolbar.Buttons(0)
MyApp.Toolbar.Buttons("Tuesday")
```

It is typical for collection indexes to begin with 0.

Each element of a collection is itself an object, as shown in the following examples:

```
Dim MyToolbarButton As Object
Set MyToolbarButton = MyApp.Toolbar.Buttons("Save")
MyAppp.Toolbar.Buttons(1).Caption = "Open"
```

The collection itself contains properties that provide you with information about the collection and methods that allow navigation within that collection:

```
Dim MyToolbarButton As Object
NumButtons% = MyApp.Toolbar.Buttons.Count
MyApp.Toolbar.Buttons.MoveNext
MyApp.Toolbar.Buttons.FindNext "Save"
For i = 1 To MyApp.Toolbar.Buttons.Count
  Set MyToolbarButton = MyApp.Toolbar.Buttons(i)
  MyToolbarButton.Caption = "Copy"
Next i
```

Predefined Objects

There are a few objects predefined for use in all macros. These are:

- Application
- Circuit
- Clipboard
- Dlg
- Err

- Msg
- Session
- Transfer

See Also “Using SmarTerm’s objects” on page 27

Oct, Oct$

Syntax **`Oct[$](number)`**

Description Returns a **`String`** containing the octal equivalent of the specified number. **`Oct$`** returns a **`String`**, whereas **`Oct`** returns a **`String`** variant. The returned string contains only the number of octal digits necessary to represent the number.

The **`number`** parameter is any numeric expression. If this parameter is **`Null`**, then **`Null`** is returned. **`Empty`** is treated as 0. The **`number`** parameter is rounded to the nearest whole number before converting to the octal equivalent.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  st$ = "The octal values are: " & crlf
  For x = 1 To 5
    y% = x * 10
    st$ = st$ & y% & " : " & Oct$(y%) & crlf
  Next x
  Session.Echo st$
End Sub
```

See Also Character and String Manipulation on page 3

OKButton

Syntax **`OKButton x,y,width,height [,.Identifier]`**

Description Creates an OK button within a dialog template. This statement can only appear within a dialog template (i.e., between the **`Begin Dialog`** and **`End Dialog`** statements). The **`OKButton`** statement accepts the following parameters:

Parameter	Description
`x`, **`y`**	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
`width`, **`height`**	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
`.Identifier`	Name by which this control can be referenced by statements in a dialog function (such as **`DlgFocus`** and **`DlgEnable`**).

If the **DefaultButton** parameter is not specified in the **Dialog** statement, the OK button will be used as the default button. In this case, the OK button can be selected by pressing Enter on a nonbutton control.

A dialog template must contain at least one **OKButton**, **CancelButton**, or **PushButton** statement (otherwise, the dialog cannot be dismissed).

Example

```
Sub Main
  Begin Dialog ButtonTemplate 17,33,104,23,"Buttons"
    OKButton 8,4,40,14,.OK
    CancelButton 56,4,40,14,.Cancel
  End Dialog
  Dim ButtonDialog As ButtonTemplate
  WhichButton = Dialog(ButtonDialog)
  If WhichButton = -1 Then
    MsgBox "OK was pressed."
  ElseIf WhichButton = 0 Then
    MsgBox "Cancel was pressed."
  End If

End Sub
```

See Also User Interaction on page 16

On Error

Syntax **On Error {Goto label | Resume Next | Goto 0}**

Description Defines the action taken when a trappable runtime error occurs. The form **On Error Goto label** causes execution to transfer to the specified label when a runtime error occurs. The form **On Error Resume Next** causes execution to continue on the line following the line that caused the error. The form **On Error Goto 0** causes any existing error trap to be removed.

If an error trap is in effect when the macro ends, then an error will be generated. An error trap is only active within the subroutine or function in which it appears. Once an error trap has gained control, appropriate action should be taken, and then control should be resumed using the **Resume** statement. The **Resume** statement resets the error handler and continues execution. If a procedure ends while an error is pending, then an error will be generated. (The **Exit Sub** or **Exit Function** statement also resets the error handler, allowing a procedure to end without displaying an error message.)

Errors within an Error Handler

If an error occurs within the error handler, then the error handler of the caller (or any procedure in the call stack) will be invoked. If there is no such error handler, then the error is fatal, causing the macro to stop executing. The following statements reset the error state (i.e., these statements turn off the fact that an error occurred):

```
Resume
Err=-1
```

The **Resume** statement forces execution to continue, either on the same line or on the line following the line that generated the error. The **Err=-1** statement allows explicit resetting of the error state so that the macro can continue normal execution without resuming at the statement that caused the error condition.

The **On Error** statement will not reset the error. Thus, if an **On Error** statement occurs within an error handler, it has the effect of changing the location of a new error handler for any new errors that may occur once the error has been reset.

Example This example shows three types of error handling. The first case simply bypasses an expected error and continues. The second case creates an error branch that jumps to a common error handling routine that processes incoming errors, clears the error (with the **Resume** statement) and resumes. The third case clears all internal error handling so that execution will stop when the next error is encountered.

```
Sub Main
  Dim x%
  a = 10000
  b = 10000
  On Error Goto Pass      'Branch to this label on error.
  Do
    x% = a * b
  Loop
Pass:
  Err = -1          'Clear error status.
  Session.Echo "Cleared error status and continued."
  On Error Goto Overflow    'Branch to new error routine on any
  x% = 1000          'subsequent errors.
  x% = a * b
  x% = a / 0
  On Error Goto 0      'Clear error branching.
  x% = a * b         'Program will stop here.
  Exit Sub          'Exit before common error routine.
Overflow:           'Beginning of common error routine.
  If Err = 6 then
    Session.Echo "Overflow Branch."
  Else
    Session.Echo Error(Err)
  End If
  Resume Next
End Sub
```

See Also Macro Control and Compilation on page 10

Open

Syntax **`Open filename$ [For mode] [Access accessmode] [lock] As [#] filenumber [Len = reclen]`**

Description Opens a file for a given mode, assigning the open file to the supplied **`filenumber`**. The **`filename$`** parameter is a string expression that contains a valid filename. The **`filenumber`** parameter is a number between 1 and 255. The **`FreeFile`** function can be used to determine an available file number. The **`mode`** parameter determines the type of operations that can be performed on that file:

File Mode	Description
`Input`	Opens an existing file for sequential input (**`filename$`** must exist). The value of **`accessmode`**, if specified, must be Read.
`Output`	Opens an existing file for sequential output, truncating its length to zero, or creates a new file. The value of **`accessmode`**, if specified, must be Write.
`Append`	Opens an existing file for sequential output, positioning the file pointer at the end of the file, or creates a new file. The value of **`accessmode`**, if specified, must be Read Write.
`Binary`	Opens an existing file for binary I/O or creates a new file. Existing binary files are never truncated in length. The value of **`accessmode`**, if specified, determines how the file can subsequently be accessed.
`Random`	Opens an existing file for record I/O or creates a new file. Existing random files are truncated only if **`accessmode`** is Write. The **`reclen`** parameter determines the record length for I/O operations.

If the **`mode`** parameter is missing, then **`Random`** is used.

The **`accessmode`** parameter determines what type of I/O operations can be performed on the file:

Access	Description
`Read`	Opens the file for reading only. This value is valid only for files opened in Binary, Random, or Input mode.
`Write`	Opens the file for writing only. This value is valid only for files opened in Binary, Random, or Output mode.
`Read Write`	Opens the file for both reading and writing. This value is valid only for files opened in Binary, Random, or Append mode.

If the **`accessmode`** parameter is not specified, the following defaults are used:

File Mode	Default Value for **`accessmode`**
`Input`	Read
`Output`	Write

File Mode	Default Value for **`accessmode`**
`Append`	Read Write
`Binary`	When the file is initially opened, access is attempted three times in the following order: 1. Read Write 2. Write 3. Read
`Random`	Same as Binary files

The **`lock`** parameter determines what access rights are granted to other processes that attempt to open the same file. The following table describes the values for **`lock`**:

Lock Value	Description
`Shared`	Other processes can read and write file. (Deny none.)
`Lock Read`	Other processes can write but not read file. (Deny read.)
`Lock Write`	Other processes can read but not write file. (Deny write.)
`Lock Read Write`	Other processes can neither read nor write file. (Exclusive.)

If **`lock`** is not specified, then the file is opened in **`Shared`** mode.

If the file does not exist and the **`lock`** parameter is specified, the file is opened twice; once to create the file and again to establish the correct sharing mode.

Files opened in **`Random`** mode are divided up into a sequence of records, each of the length specified by the **`reclen`** parameter. If this parameter is missing, then 128 is used. For files opened for sequential I/O, the **`reclen`** parameter specifies the size of the internal buffer used by the compiler when performing I/O. Larger buffers mean faster file access. For **`Binary`** files, the **`reclen`** parameter is ignored.

For files opened in **`Append`** mode, the compiler opens the file and positions the file pointer after the last character in the file. The end-of-file character, if present, is not removed.

Example

```
Sub Main
  Open "test.dat" For Output Access Write Lock Write As #2
  Close
  Open "test.dat" For Input Access Read Shared As #1
  Close
  Open "test.dat" For Append Access Write Lock Read Write as #3
  Close
  Open "test.dat" For Binary Access Read Write Shared As #4
  Close
  Open "test.dat" For Random Access Read Write Lock Read As #5
  Close
  Open "test.dat" For Input Access Read Shared As #6
```

```
  Close
  Kill "test.dat"
End Sub
```

See Also Drive, Folder, and File Access on page 4

OpenFilename$

Syntax **`OpenFilename$[([title$ [,[extensions$] [,helpfile,context]]])]`**

Description Displays a dialog that prompts the user to select from a list of files, returning the full pathname of the file the user selects or a zero-length string if the user selects Cancel. This function displays the standard file open dialog, which allows the user to select a file. It takes the following parameters:

Parameter	Description
`title$`	String specifying the title that appears in the dialog's title bar. If this parameter is omitted, then "Open" is used.
`extension$`	String specifying the available file types. If this parameter is omitted, then all files are displayed.
`helpfile`	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then **`context`** must also be specified.
`context`	Number specifying the ID of the topic within **`helpfile`** for this dialog's help. If this parameter is specified, then **`helpfile`** must also be specified.

If both the **`helpfile`** and **`context`** parameters are specified, then a Help button is added in addition to the OK and Cancel buttons. Context-sensitive help can be invoked by selecting this button or using the help key F1. Invoking help does not remove the dialog.

The **`extensions$`** parameter must be in the following format:

```
type:ext[,ext][;type:ext[,ext]]...
```

Placeholder	Description
`type`	Specifies the name of the grouping of files, such as All Files.
`ext`	Specifies a valid file extension, such as *.BAT or *.?F?.

For example, the following are valid **`extensions$`** specifications:

```
"All Files:*.*"
"Documents:*.TXT,*.DOC"
"All Files:*.*;Documents:*.TXT,*.DOC"
```

Example

```
Sub Main
  Dim f As String,s As String
  f$ = OpenFilename$("Open Picture","Text Files:*.TXT")
  If f$ <> "" Then
```

```
    Open f$ For Input As #1
    Line Input #1,s$
    Close #1
    MsgBox "First line from " & f$ & " is " & s$
  End If
End Sub
```

See Also Drive, Folder, and File Access on page 4; User Interaction on page 16

Operator Precedence (topic)

The following table shows the precedence of the operators. Operations involving operators of higher precedence occur before operations involving operators of lower precedence. When operators of equal precedence occur together, they are evaluated from left to right.

Operator	Description	Precedence
`()`	Parentheses	Highest
`^`	Exponentiation	
`-`	Unary minus	
`/, *`	Division and multiplication	
`\`	Integer division	
`Mod`	Modulo	
`+, -`	Addition and subtraction	
`&`	String concatenation	
`=, <>, >, <, <=, >=`	Relational	
`Like, Is`	String and object comparison	
`Not`	Logical negation	
`And`	Logical or binary conjunction	
`Or`	Logical or binary disjunction	
`Xor, Eqv, Imp`	Logical or binary operators	Lowest

The precedence order can be controlled using parentheses, as shown below:

```
a = 4 + 3 * 2       'a becomes 10.
a = (4 + 3) * 2     'a becomes 14.
```

Operator Precision (topic)

When numeric, binary, logical or comparison operators are used, the data type of the result is generally the same as the data type of the more precise operand. For example, adding an **`Integer`** and a **`Long`** first converts the **`Integer`** operand to a **`Long`**, then performs a long addition, overflowing only if the result cannot be contained with a **`Long`**. The order of precision is shown in the following list:

Data Type	Precision
Empty	Least precise
Boolean	
Integer	
Long	
Single	
Date	
Double	
Currency	Most precise

There are exceptions noted in the descriptions of each operator.

The rules for operand conversion are further complicated when an operator is used with variant data. In many cases, an overflow causes automatic promotion of the result to the next highest precise data type. For example, adding two **Integer** variants results in an **Integer** variant unless it overflows, in which case the result is automatically promoted to a **Long** variant.

Option Base

Syntax **Option Base {0 | 1}**

Description Sets the lower bound for array declarations. By default, the lower bound used for all array declarations is 0. This statement must appear outside of any functions or subroutines.

Example

```
Option Base 1
Sub Main
  Dim a(10)      'Contains 10 elements (not 11).
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Option Compare

Syntax **Option Compare [Binary | Text]**

Description Controls how strings are compared. When **Option Compare** is set to **Binary**, then string comparisons are case-sensitive (e.g., "A" does not equal "a"). When it is set to **Text**, string comparisons are case-insensitive (e.g., "A" is equal to "a"). The default value for **Option Compare** is **Binary**.

The **Option Compare** statement affects all string comparisons in any statements that follow the **Option Compare** statement. Additionally, the setting affects the default behavior of **Instr**, **StrComp**, and the **Like** operator. The following table shows the types of string comparisons affected by this setting:

`>`	`<`	`<>`
`<=`	`>=`	`Instr`
`StrComp`	`Like`	

The **`Option Compare`** statement must appear outside the scope of all subroutines and functions. In other words, it cannot appear within a **`Sub`** or **`Function`** block.

Example

```
Option Compare Binary
Sub CompareBinary
  a$ = "This String Contains UPPERCASE."
  b$ = "this string contains uppercase."
  If a$ = b$ Then
    MsgBox "The two strings were compared case-insensitive."
  Else
    MsgBox "The two strings were compared case-sensitive."
  End If
End Sub
Option Compare Text
Sub CompareText
  a$ = "This String Contains UPPERCASE."
  b$ = "this string contains uppercase."
  If a$ = b$ Then
    MsgBox "The two strings were compared case-insensitive."
  Else
    MsgBox "The two strings were compared case-sensitive."
  End If
End Sub
Sub Main
'!
  CompareBinary          'Calls subroutine above.
  CompareText          'Calls subroutine above.
End Sub
```

See Also Character and String Manipulation on page 3

Option CStrings

Syntax **`Option CStrings {On | Off}`**

Description Turns on or off the ability to use C-style escape sequences within strings. When **`Option CStrings On`** is in effect, the compiler treats the backslash character as an escape character when it appears within strings. An escape character is simply a special character that otherwise cannot ordinarily be typed by the computer keyboard.

Escape	Description	Equivalent Expression
`\r`	Carriage return	`Chr$(13)`
`\n`	Line Feed	`Chr$(10)`
`\a`	Bell	`Chr$(7)`
`\b`	Backspace	`Chr$(8)`

Escape	Description	Equivalent Expression
`\f`	Form Feed	`Chr$(12)`
`\t`	Tab	`Chr$(9)`
`\v`	Vertical tab	`Chr$(11)`
`\0`	Null	`Chr$(0_`
`\"`	Double quote	`""` or `Chr$(34)`
`\\`	Backslash	`Chr$(92)`
`\?`	Question mark	`?`
`\'`	Single quote	`'`
\x**hh**	Hexadecimal number	`Chr$(Val(&Hhh))`
ooo	Octal number	`Chr$(Val(&Oooo))`
anycharacter	Any character	`anycharacter`

With hexadecimal values, the compiler stops scanning for digits when it encounters a nonhexadecimal digit or two digits, whichever comes first. Similarly, with octal values, the compiler stops scanning when it encounters a nonoctal digit or three digits, whichever comes first.

When **`Option CStrings Off`** is in effect, then the backslash character has no special meaning. This is the default.

Example

```
Option CStrings On
Sub Main
  MsgBox "They said, \"Watch out for that clump of grass!\""
  MsgBox "First line.\r\nSecond line."
  MsgBox "Char A: \x41 \r\n Char B: \x42"
End Sub
```

See Also Character and String Manipulation on page 3

Option Default

Syntax **`Option Default type`**

Description Sets the default data type of variables and function return values when not otherwise specified. By default, the type of implicitly defined variables and function return values is **`Variant`**. This statement is used for backward compatibility with earlier versions of VBA where the default data type was **`Integer`**.

Note This statement must appear outside the scope of all functions and subroutines.

Currently, **`type`** can only be set to **`Integer`**.

Example

```
Option Default Integer

Function AddIntegers(a As Integer,b As Integer)
  Foo = a + b
End Function

Sub Main
  Dim a,b,result
  a = InputBox("Enter an integer:")
  b = InputBox("Enter an integer:")
  result = AddIntegers(a,b)
End Sub
```

See Also Macro Control and Compilation on page 10

Option Explicit

Syntax `Option Explicit`

Description The **`Option Explicit`** statement enforces explicit declaration of variables with **`Dim`**, **`Public`**, or **`Private`**. By default, the compiler implicitly declares variables that are used but have not been explicitly declared with **`Dim`**, **`Public`**, or **`Private`**. To avoid typing errors, use **`Option Explicit`** to prevent this behavior.

The **`Option Explicit`** statement also enforces explicit declaration of all subroutines and functions (with the **`Declare`** statement) called by other members of the macro collective. Once specified, all externally called subroutines and functions must be explicitly declared with the **`Declare`** statement.

Note Functions called by other members of the macro collective must always be declared with the **`Declare`** statement. This does not mean that you must also always use the **`Option Explicit`** statement; if you do not use **`Option Explicit`**, you can declare functions without declaring subroutines. Note, also, that not all members of the macro collective can supply subroutines and functions to the rest of the collective. See “Modules and collectives” on page 32 for more information.

See Also Declare on page 209; Macro Control and Compilation on page 10

OptionButton

Syntax `OptionButton x,y,width,height,title$ [,.Identifier]`

Description Defines an option button within a dialog template. This statement can only appear within a dialog template (i.e., between the **`Begin Dialog`** and **`End Dialog`** statements). The **`OptionButton`** statement accepts the following parameters:

Parameter	Description
x, **y**	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
width, **height**	Integer coordinates specifying the dimensions of the control in dialog units.
title$	String containing text that appears within the option button. This text may contain an ampersand character to denote an accelerator letter, such as "&Portrait" for Portrait, which can be selected by pressing the P accelerator.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**).

Accelerators are underlined, and the accelerator combination Alt+**letter** is used.

Example **See OptionGroup (statement).**

See Also User Interaction on page 16

OptionGroup

Syntax **OptionGroup .Identifier**

Description Specifies the start of a group of option buttons within a dialog template. The **.Identifier** parameter specifies the name by which the group of option buttons can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). This parameter also creates an integer variable whose value corresponds to the index of the selected option button within the group (0 is the first option button, 1 is the second option button, and so on). This variable can be accessed using the following syntax: **DialogVariable.Identifier**.

This statement can only appear within a dialog template (i.e., between the **Begin Dialog** and **End Dialog** statements).

When the dialog is created, the option button specified by **.Identifier** will be on; all other option buttons in the group will be off. When the dialog is dismissed, the **.Identifier** will contain the selected option button.

Example

```
Sub Main
  Begin Dialog PrintTemplate 16,31,128,65,"Print"
    GroupBox 8,8,64,52,"Orientation",.Junk
    OptionGroup .Orientation
      OptionButton 16,20,37,8,"Portrait",.Portrait
      OptionButton 16,32,51,8,"Landscape",.Landscape
      OptionButton 16,44,49,8,"Don't Care",.DontCare
    OKButton 80,8,40,14
  End Dialog
  Dim PrintDialog As PrintTemplate
  Dialog PrintDialog
End Sub
```

See Also User Interaction on page 16

Or

Syntax **`result = expression1 Or expression2`**

Description Performs a logical or binary disjunction on two expressions. If both expressions are either **`Boolean`**, **`Boolean`** variants, or **`Null`** variants, then a logical disjunction is performed as follows:

Expression One	Expression Two	Result
True	True	True
True	False	True
True	Null	True
False	True	True
False	False	False
False	Null	Null
Null	True	True
Null	False	Null
Null	Null	Null

Binary Disjunction

If the two expressions are **`Integer`**, then a binary disjunction is performed, returning an **`Integer`** result. All other numeric types (including **`Empty`** variants) are converted to **`Long`** and a binary disjunction is then performed, returning a **`Long`** result.

Binary disjunction forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions according to the following table:

Bit in Expression One	Bit in Expression Two	Result
1	1	1
0	1	1
1	0	1
0	0	0

Examples This first example shows the use of logical Or.

```
Dim s$ As String
s$ = InputBox$("Enter a string.")
If s$ = "" Or Mid$(s$,1,1) = "A" Then
  s$ = LCase$(s$)
End If
```

This second example shows the use of binary Or.

```
Dim w As Integer
TryAgain:
  s$ = InputBox$("Enter a hex number (four digits max).")
  If Mid$(s$,1,1) <> "&" Then
    s$ = "&H" & s$
  End If
  If Not IsNumeric(s$) Then Goto TryAgain
  w = CInt(s$)
  MsgBox "Your number is &H" & Hex$(w)
  w = w Or &H8000
  MsgBox "Your number with the high bit set is &H" & Hex$(w)
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

P

Picture

Syntax **`Picture x,y,width,height,PictureName$,PictureType [,[.Identifier] [,style]]`**

Description Creates a picture control in a dialog template. Picture controls are used for the display of graphics images only. The user cannot interact with these controls. The **`Picture`** statement accepts the following parameters:

Parameter	Description
`x`, **`y`**	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
`width`, **`height`**	Integer coordinates specifying the dimensions of the control in dialog units.
`PictureName$`	String containing the name of the picture. If **`PictureType`** is 0, then this name specifies the name of the file containing the image. If **`PictureType`** is 10, then **`PictureName$`** specifies the name of the image within the resource of the picture library. If **`PictureName$`** is empty, then no picture will be associated with the control. A picture can later be placed into the picture control using the **`DlgSetPicture`** statement.
`PictureType`	Integer specifying the source for the image. The following sources are supported: 0 The image is contained in a file on disk. 10 The image is contained in a picture library as specified by the **`PicName$`** parameter on the Begin Dialog statement.
`.Identifier`	Name by which this control can be referenced by statements in a dialog function (such as **`DlgFocus`** and **`DlgEnable`**). If omitted, then the first two words of **`PictureName$`** are used.

Parameter	Description
`style`	Specifies whether the picture is drawn within a 3D frame. It can be either of the following values: 0 Draw the picture control with a normal frame. 1 Draw the picture control with a 3D frame. If this parameter is omitted, then the picture control is drawn with a normal frame.

The picture control extracts the actual image from either a disk file or a picture library. In the case of bitmaps, both 2- and 16-color bitmaps are supported. In the case of WMFs, the compiler supports the Placeable Windows Metafile.

If **`PictureName$`** is a zero-length string, then the picture is removed from the picture control, freeing any memory associated with that picture.

Picture controls can contain either a bitmap or a WMF (Windows metafile). When extracting images from a picture library, the compiler assumes that the resource type for metafiles is 256. Picture libraries are implemented as DLLs.

Examples This first example shows how to use a picture from a file.

```
Sub Main
  Begin Dialog LogoDialogTemplate 16,32,288,76,"Introduction"
    OKButton 240,8,40,14
    Picture 8,8,224,64,"c:\bitmaps\logo.bmp",0,.Logo
  End Dialog
  Dim LogoDialog As LogoDialogTemplate
  Dialog LogoDialog
End Sub
```

This second example shows how to use a picture from a picture library with a 3D frame.

```
Sub Main
  Begin Dialog LogoDialogTemplate _
    16,31,288,76,"Introduction",,"pictures.dll"
    OKButton 240,8,40,14
    Picture 8,8,224,64,"CompanyLogo",10,.Logo,1
  End Dialog
  Dim LogoDialog As LogoDialogTemplate
  Dialog LogoDialog
End Sub
```

See Also User Interaction on page 16

PictureButton

Syntax **`PictureButton x,y,width,height,PictureName$,PictureType [,.Identifier]`**

Description Creates a picture button control in a dialog template. Picture button controls behave very much like push button controls. Visually, picture buttons are different from push buttons in that they contain a graphic image imported either from a file or from a picture library. The **`PictureButton`** statement accepts the following parameters:

Parameter	Description
`x`, **`y`**	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
`width`, **`height`**	Integer coordinates specifying the dimensions of the control in dialog units.
`PictureName$`	String containing the name of the picture. If **`PictureType`** is 0, then this name specifies the name of the file containing the image. If **`PictureType`** is 10, then **`PictureName$`** specifies the name of the image within the resource of the picture library. If **`PictureName$`** is empty, then no picture will be associated with the control. A picture can later be placed into the picture control using the **`DlgSetPicture`** statement.
`PictureType`	Integer specifying the source for the image. The following sources are supported: • The image is contained in a file on disk. • The image is contained in a picture library as specified by the **`PicName$`** parameter on the Begin Dialog statement.
`.Identifier`	Name by which this control can be referenced by statements in a dialog function (such as **`DlgFocus`** and **`DlgEnable`**).

The picture button control extracts the actual image from either a disk file or a picture library, depending on the value of **`PictureType`**.

If **`PictureName$`** is a zero-length string, then the picture is removed from the picture button control, freeing any memory associated with that picture.

Picture controls can contain either a bitmap or a WMF (Windows metafile). When extracting images from a picture library, the compiler assumes that the resource type for metafiles is 256. Picture libraries are implemented as DLLs.

Examples This first example shows how to use a picture from a file.

```
Sub Main
  Begin Dialog LogoDialogTemplate 16,32,288,76,"Introduction"
    OKButton 240,8,40,14
    PictureButton 8,4,224,64,"c:\bitmaps\logo.bmp",0,.Logo
  End Dialog
  Dim LogoDialog As LogoDialogTemplate
  Dialog LogoDialog
End Sub
'This second example shows how to use a picture from a picture
'library.

Sub Main
```

```
  Begin Dialog LogoDialogTemplate 16,31,288,76,"Introduction",,"pictures.dll"
    OKButton 240,8,40,14
    PictureButton 8,4,224,64,"CompanyLogo",10,.Logo
  End Dialog
  Dim LogoDialog As LogoDialogTemplate
  Dialog LogoDialog
End Sub
```

See Also User Interaction on page 16

Pmt

Syntax **`Pmt(rate, nper, pv, fv, due)`**

Description Returns the payment for an annuity based on periodic fixed payments and a constant rate of interest. An annuity is a series of fixed payments made to an insurance company or other investment company over a period of time. Examples of annuities are mortgages and monthly savings plans. The **`Pmt`** function requires the following named parameters:

Parameter	Description
`rate`	Double representing the interest rate per period. If the periods are given in months, be sure to normalize annual rates by dividing them by 12.
`Nper`	Double representing the total number of payments in the annuity.
`Pv`	Double representing the present value of your annuity. In the case of a loan, the present value would be the amount of the loan.
`Fv`	Double representing the future value of your annuity. In the case of a loan, the future value would be 0.
`Due`	Integer indicating when payments are due for each payment period. A 0 specifies payment at the end of each period, whereas a 1 specifies payment at the start of each period.

The **`rate`** and **`nper`** parameters must be expressed in the same units. If **`rate`** is expressed in months, then **`nper`** must also be expressed in months.

Positive numbers represent cash received, whereas negative numbers represent cash paid out.

Example This example calculates the payment necessary to repay a $1,000.00 loan over 36 months at an annual rate of 10%. Payments are due at the beginning of the period.

```
Sub Main
  x = Pmt((.1/12),36,1000.00,0,1)
  mesg = "The payment to amortize $1,000 over 36 months @ 10% is: "
  Session.Echo mesg & Format(x,"Currency")
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

PopUpMenu

Syntax **`PopUpMenu(MenuList$())`**

Description Displays a PopUp menu on the SmarTerm display screen at the point where the mouse cursor currently resides. Returns a numeric value corresponding to the menu selection.

```
Example:
Sub Main
'!
Dim RetVal as Integer
Dim MenuList$(3)
MenuList$(0)="Menu Option 1"
MenuList$(1)="Menu Option 2"
MenuList$(2)="Menu Option 3"
MenuList$(3)="Menu Option 4"
RetVal=PopUpMenu(MenuList$)
End Sub
```

PPmt

Syntax **`PPmt(rate, per, nper, pv, fv, due)`**

Description Calculates the principal payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate. An annuity is a series of fixed payments made to an insurance company or other investment company over a period of time. Examples of annuities are mortgages and monthly savings plans. The **`PPmt`** function requires the following named parameters:

Parameter	Description
`rate`	Double representing the interest rate per period.
`Per`	Double representing the number of payment periods. The **`per`** parameter can be no less than 1 and no greater than **`nper`**.
`Nper`	Double representing the total number of payments in your annuity.
`Pv`	Double representing the present value of your annuity. In the case of a loan, the present value would be the amount of the loan.
`Fv`	Double representing the future value of your annuity. In the case of a loan, the future value would be 0.
`Due`	Integer indicating when payments are due. If this parameter is 0, then payments are due at the end of each period; if it is 1, then payments are due at the start of each period.

`The rate` and **`nper`** parameters must be in the same units to calculate correctly. If **`rate`** is expressed in months, then **`nper`** must also be expressed in months.

Negative values represent payments paid out, whereas positive values represent payments received.

Example This example calculates the principal paid during each year on a loan of $1,000.00 with an annual rate of 10% for a period of 10 years. The result is displayed as a table containing the following information: payment, principal payment, principal balance.

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  pay = Pmt(.1,10,1000.00,0,1)
  mesg = "Amortization table for 1,000" & crlf & _
    "at 10% annually for"
  mesg = mesg & " 10 years: " & crlf & crlf
  bal = 1000.00
  For per = 1 to 10
    prn = PPmt(.1,per,10,1000,0,0)
    bal = bal + prn
    mesg = mesg & Format(pay,"Currency") & "  " & _
      Format$(Prn,"Currency")
    mesg = mesg & "  " & Format(bal,"Currency") & crlf
  Next per
  Session.Echo mesg
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

Print

Syntax `Print [[{Spc(n) | Tab(n)}][expressionlist][{; | ,}]]`

Description Prints data to an output device. The following table describes how data of different types is written:

Data Type	Description
String	Printed in its literal form, with no enclosing quotes.
Any numeric type	Printed with an initial space reserved for the sign (space = positive). Additionally, there is a space following each number.
Boolean	Printed as "True" or "False". These keywords are translated as appropriate according to your system’s locale.
Date	Printed using the short date format. If either the date or time component is missing, only the provided portion is printed (this is consistent with the "general date" format understood by the Format/Format$ functions).
Empty	Nothing is printed

Data Type	Description
Null	Prints "null". This keyword is translated as appropriate according to your system’s locale.
User-defined errors	User-defined errors are printed to files as "Error **`code`**", where **`code`** is the value of the user-defined error. The word "Error" is not translated. The "Error" keyword is translated as appropriate according to your system’s locale.
Object	For any object type, the compiler retrieves the default property of that object and prints this value using the above rules.

Each expression in **`expressionlist`** is separated with either a comma (,) or a semicolon (;). A comma means that the next expression is output in the next print zone. A semicolon means that the next expression is output immediately after the current expression. Print zones are defined every 14 spaces.

If the last expression in the list is not followed by a comma or a semicolon, then a carriage return is printed to the file. If the last expression ends with a semicolon, no carriage return is printed; the next **`Print`** statement will output information immediately following the expression. If the last expression in the list ends with a comma, the file pointer is positioned at the start of the next print zone on the current line.

The **`Tab`** and **`Spc`** functions provide additional control over the column position. The **`Tab`** function moves the file position to the specified column, whereas the **`Spc`** function outputs the specified number of spaces**.**

Note Null characters **`Chr$(0)`** within strings are translated to spaces when printing to the Viewport window. When printing to files, this translation is not performed.

This statement writes data to a viewport window.

If no viewport window is open, then the statement is ignored. Printing information to a viewport window is a convenient way to output debugging information. To open a viewport window, use the following statement:

```
Viewport.Open
```

Examples

```
Sub Main
  i% = 10
  s$ = "This is a test."
  Print "The value of i=";i%,"the value of s=";s$
  'This example prints the value of i% in print zone
  '1 and s$ in print zone 3.
  Print i%,,s$
  'This example prints the value of i% and s$
  'separated by 10 spaces.
  Print i%;Spc(10);s$
  'This example prints the value of i in column 1 and s$ in
  'column 30.
```

```
  Print i%;Tab(30);s$
  'This example prints the value of i% and s$.
  Print i%;s$,
  Print 67
End Sub
```

See Also Drive, Folder, and File Access on page 4

Print#

Syntax **`Print #filenumber, [[{Spc(n) | Tab(n)}][expressionlist][{;|,}]]`**

Description Writes data to a sequential disk file. The **`filenumber`** parameter is a number that is used to refer to the open file—the number passed to the **`Open`** statement. The following table describes how data of different types is written:

Data Type	Description
String	Printed in its literal form, with no enclosing quotes.
Any numeric type	Printed with an initial space reserved for the sign (space = positive). Additionally, there is a space following each number.
Boolean	Printed as "True" or "False". These keywords are translated as appropriate according to your system’s locale.
Date	Printed using the short date format. If either the date or time component is missing, only the provided portion is printed (this is consistent with the "general date" format understood by the Format/Format$ functions).
Empty	Nothing is printed
Null	Prints "null". This keyword is translated as appropriate according to your system’s locale.
User-defined errors	User-defined errors are printed to files as "**`Error code`**", where **`code`** is the value of the user-defined error. The word "Error" is not translated. The "**`Error`**" keyword is translated as appropriate according to your system’s locale.
Object	For any object type, the compiler retrieves the default property of that object and prints this value using the above rules.

Each expression in expressionlist is separated with either a comma (,) or a semicolon (;). A comma means that the next expression is output in the next print zone. A semicolon means that the next expression is output immediately after the current expression. Print zones are defined every 14 spaces.

If the last expression in the list is not followed by a comma or a semicolon, then an end-of-line is printed to the file. If the last expression ends with a semicolon, no end-of-line is printed; the next **`Print`** statement will output information immediately following the expression. If the last expression in the list ends with a comma, the file pointer is positioned at the start of the next print zone on the current line.

The **Write** statement always outputs information ending with an end-of-line. Thus, if a **Print** statement is followed by a **Write** statement, the file pointer is positioned on a new line.

The **Print** statement can only be used with files that are opened in **Output** or **Append** mode.

The **Tab** and **Spc** functions provide additional control over the file position. The **Tab** function moves the file position to the specified column, whereas the **Spc** function outputs the specified number of spaces.

In order to correctly read the data using the **Input#** statement, you should write the data using the **Write** statement.

Examples

```
Sub Main
  'This example opens a file and prints some data.
  Open "test.dat" For Output As #1
  i% = 10
  s$ = "This is a test."
  Print #1,"The value of i=";i%,"the value of s=";s$
  'This example prints the value of i% in print zone 1 and
  's$ in print zone 3.
  Print #1,i%,,s$
  'This example prints the value of i% and s$ separated by
  'ten spaces.
  Print #1,i%;Spc(10);s$
  'This example prints the value of i in column 1 and s$ in
  'column 30.
  Print #1,i%;Tab(30);s$
  'This example prints the value of i% and s$.
  Print #1,i%;s$,
  Print #1,67
  Close #1
  Kill "test.dat"
End Sub
```

See Also Drive, Folder, and File Access on page 4

Private

Syntax **Private name [(subscripts)] [As type] [,name [(subscripts)] [As type]]...**

Description Declares a list of private variables and their corresponding types and sizes. Private variables are global to every **Sub** and **Function** within the currently executing macro. If a type-declaration character is used when specifying name (such as **%**, **@**, **&**, **$**, or **!**), the optional [**As type**] expression is not allowed. For example, the following are allowed:

```
Private foo As Integer
Private foo%
```

The **subscripts** parameter allows the declaration of arrays. This parameter uses the following syntax:

```
[lower To] upper [,[lower To] upper]...
```

The **`lower`** and **`upper`** parameters are integers specifying the lower and upper bounds of the array. If **`lower`** is not specified, then the lower bound as specified by **`Option Base`** is used (or 1 if no **`Option Base`** statement has been encountered). Up to 60 array dimensions are allowed. The total size of an array (not counting space for strings) is limited to 64K. Dynamic arrays are declared by not specifying any bounds:

```
Private a()
```

The **`type`** parameter specifies the type of the data item being declared. It can be any of the following data types: **`String`**, **`Integer`**, **`Long`**, **`Single`**, **`Double`**, **`Currency`**, **`Object`**, data object, built-in data type, or any user-defined data type.

If a variable is seen that has not been explicitly declared with either **`Dim`**, **`Public`**, or **`Private`**, then it will be implicitly declared local to the routine in which it is used.

Fixed-Length Strings

Fixed-length strings are declared by adding a length to the **`String`** type-declaration character:

```
Private name As String * length
```

where **`length`** is a literal number specifying the string's length.

Initial Values

All declared variables are given initial values, as described in the following table:

Data Type	Initial Value
Integer	0
Long	0
Double	0.0
Single	0.0
Currency	0.0
Object	Nothing
Date	December 31, 1899 00:00:00
Boolean	False
Variant	Empty
String	"" (zero-length string)
User-defined type	Structure elements are given the default values listed above.
Arrays	Array elements are given the default values listed above.

Example **`See Public (statement).`**

See Also Macro Control and Compilation on page 10

Public

Syntax `Public name [(subscripts)] [As type] [,name [(subscripts)] [As type]]...`

Description Declares a list of public variables and their corresponding types and sizes. Public variables are global to all **Sub**s and **Function**s in all macros. If a type-declaration character is used when specifying name (such as **%**, **@**, **&**, **$**, or **!**), the optional [**As type**] expression is not allowed. For example, the following are allowed:

```
Public foo As integer
Public foo%
```

The **subscripts** parameter allows the declaration of arrays. This parameter uses the following syntax:

```
[lower To] upper [,[lower To] upper]...
```

The **lower** and **upper** parameters are integers specifying the lower and upper bounds of the array. If **lower** is not specified, then the lower bound as specified by **Option Base** is used (or 1 if no **Option Base** statement has been encountered). Up to 60 array dimensions are allowed. The total size of an array (not counting space for strings) is limited to 64K. Dynamic arrays are declared by not specifying any bounds:

```
Public a()
```

The **type** parameter specifies the type of the data item being declared. It can be any of the following data types: **String**, **Integer**, **Long**, **Single**, **Double**, **Currency**, **Object**, data object, built-in data type, or any user-defined data type.

If a variable is seen that has not been explicitly declared with either **Dim**, **Public**, or **Private**, then it will be implicitly declared local to the routine in which it is used.

For compatibility, the keyword **Global** is also supported. It has the same meaning as **Public**.

Fixed-Length Strings

Fixed-length strings are declared by adding a length to the **String** type-declaration character:

```
Public name As String * length
```

where **length** is a literal number specifying the string's length.

All declared variables are given initial values, as described in the following table:

Data Type	Initial Value
Integer	0
Long	0
Double	0.0
Single	0.0
Currency	0.0
Date	December 31, 1899 00:00:00
Object	Nothing
Boolean	False
Variant	Empty
String	"" (zero-length string)
User-defined type	Structure elements are given the default values listed above.
Arrays	Array elements are given the default values listed above.

Sharing Variables

When sharing variables, you must ensure that the declarations of the shared variables are the same in each macro that uses those variables. If the public variable being shared is a user-defined structure, then the structure definitions must be exactly the same.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Public x#, ar#
Sub Area()
  ar# = (x# ^ 2) * Pi
End Sub

Sub Main
  mesg = "The area of the ten circles are:" & crlf
  For x# = 1 To 10
    Area
    mesg = mesg & x# & ": " & ar# & Basic.Eoln$
  Next x#
  Session.Echo mesg
End Sub
```

See Also Macro Control and Compilation on page 10

PushButton

Syntax `PushButton x,y,width,height,title$ [,.Identifier]`

Description Defines a push button within a dialog template. Choosing a push button causes the dialog to close (unless the dialog function redefines this behavior). This statement can only appear within a dialog template (i.e., between the **`Begin Dialog`** and **`End Dialog`** statements).

The **PushButton** statement accepts the following parameters:

Parameter	Description
x, **y**	Integer coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog.
width, **height**	Integer coordinates specifying the dimensions of the control in dialog units.
title$	String containing the text that appears within the push button. This text may contain an ampersand character to denote an accelerator letter, such as **"&Save"** for Save.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**).

If a push button is the default button, it can be selected by pressing Enter on a nonbutton control.

A dialog template must contain at least one **OKButton**, **CancelButton**, or **PushButton** statement (otherwise, the dialog cannot be dismissed).

Accelerators are underlined, and the accelerator combination Alt+letter is used.

Example

```
Sub Main
  Begin Dialog ButtonTemplate 17,33,104,84,"Buttons"
    OKButton 8,4,40,14,.OK
    CancelButton 8,24,40,14,.Cancel
    PushButton 8,44,40,14,"1",.Button1
    PushButton 8,64,40,14,"2",.Button2
    PushButton 56,4,40,14,"3",.Button3
    PushButton 56,24,40,14,"4",.Button4
    PushButton 56,44,40,14,"5",.Button5
    PushButton 56,64,40,14,"6",.Button6
  End Dialog
  Dim ButtonDialog As ButtonTemplate
  WhichButton% = Dialog(ButtonDialog)
  MsgBox "You pushed button " & WhichButton%
End Sub
```

See Also User Interaction on page 16

Put

Syntax **Put [#]filenumber, [recordnumber], variable**

Description Writes data from the specified variable to a **Random** or **Binary** file. The **Put** statement accepts the following parameters:

Parameter	Description
`filenumber`	Integer representing the file to be written to. This is the same value as returned by the Open statement.
`Recordnumber`	Long specifying which record is to be written to the file. For Binary files, this number represents the first byte to be written starting with the beginning of the file (the first byte is 1). For Random files, this number represents the record number starting with the beginning of the file (the first record is 1). This value ranges from 1 to 2147483647. If the **`recordnumber`** parameter is omitted, the next record is written to the file (if no records have been written yet, then the first record in the file is written). When **`recordnumber`** is omitted, the commas must still appear, as in the following example: **`Put #1,,recvar`** If **`recordlength`** is specified, it overrides any previous change in file position specified with the Seek statement.

The **`variable`** parameter is the name of any variable of any of the following types:

Variable Type	File Storage Description
Integer	2 bytes are written to the file.
Long	4 bytes are written to the file.
String (variable-length)	In Binary files, variable-length strings are written by first determining the specified string variable's length, then writing that many bytes to a file. In Random files, variable-length strings are written by first writing a 2-byte length, then writing that many characters to the file.
String (fixed-length)	Fixed-length strings are written to Random and Binary files in the same way: the number of characters equal to the string's declared length are written.
Double	8 bytes are written to the file (IEEE format),
Single	4 bytes are written to the file (IEEE format).
Date	8 bytes are written to the file (IEEE double format).
Boolean	2 bytes are written to the file (either –1 for True or 0 for False).
Variant	A 2-byte VarType is written to the file followed by the data as described above. With variants of type 10 (user-defined errors), the 2-byte VarType is followed by a 2-byte unsigned integer (the error value), which is then followed by 2 additional bytes of information. The exception is with strings, which are always preceded by a 2-byte string length.

Variable Type	File Storage Description
User-defined types	Each member of a user-defined data type is written individually. In Binary files, variable-length strings within user-defined types are written by first writing a 2-byte length followed by the string's content. This storage is different than variable-length strings outside of user-defined types. When writing user-defined types, the record length must be greater than or equal to the combined size of each element within the data type.
Arrays	Arrays cannot be written to a file using the **`Put`** statement.
Objects	Object variables cannot be written to a file using the **`Put`** statement.

With **`Random`** files, a runtime error will occur if the length of the data being written exceeds the record length (specified as the **`reclen`** parameter with the **`Open`** statement). If the length of the data being written is less than the record length, the entire record is written along with padding (whatever data happens to be in the I/O buffer at that time). With **`Binary`** files, the data elements are written contiguously: they are never separated with padding.

Example

```
Sub Main
  Open "test.dat" For Random Access Write As #1
  For x = 1 To 10
    r% = x * 10
    Put #1,x,r%
  Next x
  Close
  Open "test.dat" For Random Access Read As #1
  For x = 1 To 10
    Get #1,x,r%
    mesg = mesg & "Record " & x & " is: " & r% & Basic.Eoln$
  Next x
  Session.Echo mesg
  Close
  Kill "test.dat"
End Sub
```

See Also Drive, Folder, and File Access on page 4

Pv

Syntax **`Pv(rate, nper, pmt, fv, due)`**

Description Calculates the present value of an annuity based on future periodic fixed payments and a constant rate of interest. The **`Pv`** function requires the following named parameters:

Parameter	Description
`rate`	Double representing the interest rate per period. When used with monthly payments, be sure to normalize annual percentage rates by dividing them by 12.
`Nper`	Double representing the total number of payments in the annuity.
`Pmt`	Double representing the amount of each payment per period.
`Fv`	Double representing the future value of the annuity after the last payment has been made. In the case of a loan, the future value would be 0.
`Due`	Integer indicating when the payments are due for each payment period. A 0 specifies payment at the end of each period, whereas a 1 specifies payment at the start of each period.

The **`rate`** and **`nper`** parameters must be expressed in the same units. If **`rate`** is expressed in months, then **`nper`** must also be expressed in months.

Positive numbers represent cash received, whereas negative numbers represent cash paid out.

Example This example demonstrates the present value (the amount you'd have to pay now) for a $100,000 annuity that pays an annual income of $5,000 over 20 years at an annual interest rate of 10%.

```
Sub Main
  pval = Pv(.1,20,-5000,100000,1)
  Session.Echo "The present value is: " & Format(pval,"Currency")
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

R

Random

Syntax `Random(min,max)`

Description Returns a **`Long`** value greater than or equal to **`min`** and less than or equal to **`max`**. Both the **`min`** and **`max`** parameters are rounded to **`Long`**. A runtime error is generated if **`min`** is greater than **`max`**.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Randomize          'Start with new random seed.
  For x = 1 To 10
    y = Random(0,100)    'Generate numbers.
    mesg = mesg & y & crlf
  Next x
  Session.Echo "Ten numbers for the lottery: " & crlf & mesg
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

Randomize

Syntax `Randomize [number]`

Description Initializes the random number generator with a new seed. If **`number`** is not specified, then the current value of the system clock is used.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Randomize          'Start with new random seed.
  For x = 1 To 10
    y = Random(0,100)    'Generate numbers.
    mesg = mesg + Str(y) + crlf
  Next x
  Session.Echo "Ten numbers for the lottery: " & crlf & mesg
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

Rate

Syntax **`Rate(nper, pmt, pv, fv, due, guess)`**

Description Returns the rate of interest for each period of an annuity. An annuity is a series of fixed payments made to an insurance company or other investment company over a period of time. Examples of annuities are mortgages and monthly savings plans. The **`Rate`** function requires the following named parameters:

Parameter	Description
`nper`	Double representing the total number of payments in the annuity.
`Pmt`	Double representing the amount of each payment per period.
`Pv`	Double representing the present value of your annuity. In a loan situation, the present value would be the amount of the loan.
`Fv`	Double representing the future value of the annuity after the last payment has been made. In the case of a loan, the future value would be zero.
`Due`	Integer specifying when the payments are due for each payment period. A 0 indicates payment at the end of each period, whereas a 1 indicates payment at the start of each period.
`Guess`	Double specifying a guess as to the value the Rate function will return. The most common guess is .1 (10 percent).

Positive numbers represent cash received, whereas negative values represent cash paid out.

The value of **`Rate`** is found by iteration. It starts with the value of **`guess`** and cycles through the calculation adjusting **`guess`** until the result is accurate within 0.00001 percent. After 20 tries, if a result cannot be found, **`Rate`** fails, and the user must pick a better guess.

Example This example calculates the rate of interest necessary to save $8,000 by paying $200 each year for 48 years. The guess rate is 10%.

```
Sub Main
  r# = Rate(48,-200,8000,0,1,.1)
  Session.Echo "The rate required is: " & Format(r#,"Percent")
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

ReadIni$

Syntax **`ReadIni$(section$,item$[,filename$])`**

Description Returns a **`String`** containing the specified item from an INI file. The **`ReadIni$`** function takes the following parameters:

Parameter	Description
`section$`	String specifying the section that contains the desired variable, such as "windows". Section names are specified without the enclosing brackets.
`item$`	String specifying the item whose value is to be retrieved.
`Filename$`	String containing the name of the INI file to read.

The maximum length of a string returned by this function is 4096 characters.

If the name of the INI file is not specified, then win.ini is assumed.

If the **`filename$`** parameter does not include a path, then this statement looks for INI files in the Windows directory.

See Also Drive, Folder, and File Access on page 4

ReadIniSection

Syntax **`ReadIniSection section$,ArrayOfItems()[,filename$]`**

Description Fills an array with the item names from a given section of the specified INI file. The **`ReadIniSection`** statement takes the following parameters:

Parameter	Description
`section$`	String specifying the section that contains the desired variables, such as "windows". Section names are specified without the enclosing brackets.
`ArrayOfItems()`	Specifies either a zero- or a one-dimensioned array of strings or variants. The array can be either dynamic or fixed. If **`ArrayOfItems()`** is dynamic, then it will be redimensioned to exactly hold the new number of elements. If there are no elements, then the array will be redimensioned to contain no dimensions. You can use the LBound, UBound, and ArrayDims functions to determine the number and size of the new array's dimensions. If the array is fixed, each array element is first erased, then the new elements are placed into the array. If there are fewer elements than will fit in the array, then the remaining elements are initialized to zero-length strings (for string arrays) or empty (for variant arrays). A runtime error results if the array is too small to hold the new elements.
`Filename$`	String containing the name of an INI file.

On return, the `ArrayOfItems()` parameter will contain one array element for each variable in the specified INI section. The maximum combined length of all the entry names returned by this function is limited to 32K.

If the name of the INI file is not specified, then win.ini is assumed.

If the `filename$` parameter does not include a path, then this statement looks for INI files in the Windows directory.

Example

```
Sub Main
  Dim items() As String
  ReadIniSection "windows",items$
  Session.Echo "INI Items:<CR><LF>"
  For i=0 to UBound(items$)
    Session.Echo item$(i) & "<CR><LF>"
  Next i
End Sub
```

See Also Drive, Folder, and File Access on page 4

Redim

Syntax

```
Redim [Preserve] variablename ([subscriptRange]) [As type],...
```

Description Redimensions an array, specifying a new upper and lower bound for each dimension of the array. The `variablename` parameter specifies the name of an existing array (previously declared using the `Dim` statement) or the name of a new array variable. If the array variable already exists, then it must previously have been declared with the `Dim` statement with no dimensions, as shown in the following example:

```
Dim a$()    'Dynamic array of strings (no dimensions yet)
```

Dynamic arrays can be redimensioned any number of times.

The `subscriptRange` parameter specifies the new upper and lower bounds for each dimension of the array using the following syntax:

```
[lower To] upper [,[lower To] upper]...
```

If `subscriptRange` is not specified, then the array is redimensioned to have no elements.

If `lower` is not specified, then 0 is used (or the value set using the `Option Base` statement). A runtime error is generated if `lower` is less than `upper`. Array dimensions must be within the following range:

```
–32768 <= lower <= upper <= 32767
```

The `type` parameter can be used to specify the array element type. Arrays can be declared using any fundamental data type, user-defined data types, and objects.

Redimensioning an array erases all elements of that array unless the **Preserve** keyword is specified. When this keyword is specified, existing data in the array is preserved where possible. If the number of elements in an array dimension is increased, the new elements are initialized to 0 (or empty string). If the number of elements in an array dimension is decreased, then the extra elements will be deleted. If the **Preserve** keyword is specified, then the number of dimensions of the array being redimensioned must either be zero or the same as the new number of dimensions.

Example

```
Sub Main
  Dim fl$()
  FileList fl$,"*.*"
  count = Ubound(fl$)
  Redim nl$(Lbound(fl$) To Ubound(fl$))
  For x = 1 to count
    nl$(x) = fl(x)
  Next x
  Session.Echo "The last element of the new array is: " & nl$(count)
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Rem

Syntax `Rem text`

Description Causes the compiler to skip all characters on that line.

Example

```
Sub Main
  Rem This is a line of comments that serves to illustrate the
  Rem workings of the code. You can insert comments to make it
  Rem more readable and maintainable in the future.
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6; Macro Control and Compilation on page 10

Reset

Syntax `Reset`

Description Closes all open files, writing out all I/O buffers.

Example

```
Sub Main
  Open "test.dat" for Output Access Write as # 1
  Reset
  Kill "test.dat"
  If FileExists("test.dat") Then
    Session.Echo "The file was not deleted."
  Else
    Session.Echo "The file was deleted."
  End If
End Sub
```

See Also Drive, Folder, and File Access on page 4

Resume

Syntax **`Resume {[0] | Next | label}`**

Description Ends an error handler and continues execution.

The form **`Resume 0`** (or simply **`Resume`** by itself) causes execution to continue with the statement that caused the error.

The form **`Resume Next`** causes execution to continue with the statement following the statement that caused the error.

The form **`Resume label`** causes execution to continue at the specified label.

The **`Resume`** statement resets the error state. This means that, after executing this statement, new errors can be generated and trapped as normal.

Example This example accepts two integers from the user and attempts to multiply the numbers together. If either number is larger than an integer, the program processes an error routine and then continues program execution at a specific section using **`Resume <label>`**. Another error trap is then set using **`Resume Nex`**t. The new error trap will clear any previous error branching and also tell the program to continue execution of the program even if an error is encountered.

```
Sub Main
  Dim a%, b%, x%
Again:
  On Error Goto Overflow
  a% = InputBox("Enter 1st integer to multiply","Enter Number")
  b% = InputBox("Enter 2nd integer to multiply","Enter Number")
  On Error Resume Next  'Continue program execution at
    x% = a% * b%    'next line if an error occurs.
  if err = 0 then
    Session.Echo x%
  else
    Session.Echo a% & " * " & b% & " cause an overflow!"
  end if
  Exit Sub
Overflow:          'Error handler.
  Session.Echo "You've entered a noninteger value. Try again!"
  Resume Again
End Sub
```

See Also Macro Control and Compilation on page 10

Return

Syntax **`Return`**

Description Transfers execution control to the statement following the most recent **GoSub**. A runtime error results if a **Return** statement is encountered without a corresponding **GoSub** statement.

Example

```
Sub Main
  GoSub SubTrue
  Session.Echo "The Main routine continues here."
  Exit Sub
SubTrue:
  Session.Echo "This message is generated in the subroutine."
  Return
  Exit Sub
End Sub
```

See Also Macro Control and Compilation on page 10

Right, Right$, RightB, RightB$

Syntax

```
Right[$](string, length)
RightB[$](string, length)
```

Description Returns the rightmost **length** characters (for **Right** and **Right$**) or bytes (for **RightB** and **RightB$**) from a specified string. The **Right$** and **RightB$** functions return a **String**, whereas the **Right** and **RightB** functions return a **String** variant. These functions take the following named parameters:

Parameter	Description
string	String from which characters are returned. A runtime error is generated if **string** is null.
Length	Integer specifying the number of characters or bytes to return. If **length** is greater than or equal to the length of the string, then the entire string is returned. If **length** is 0, then a zero-length string is returned.

The **RightB** and **RightB$** functions are used to return byte data from strings containing byte data.

Example

```
Sub Main
  lname$ = "WILLIAMS"
  x = Len(lname$)
  rest$ = Right$(lname$,x - 1)
  fl$ =  Left$(lname$,1)
  lname$ = fl$ & LCase$(rest$)
  Session.Echo "The converted name is: " & lname$
End Sub
```

See Also Character and String Manipulation on page 3

RmDir

Syntax

```
RmDir path
```

Description Removes the directory specified by the **String** contained in **path**.

Note Removing a directory that is the current directory on that drive causes unpredictable side effects. For example, consider the following statements:

```
MkDir "Z:\JUNK"
ChDir "Z:\JUNK"
RmDir "Z:\JUNK"
```

If drive Z is a network drive, then some networks will delete the directory and unmap the drive without generating a macro error. If drive Z is a local drive, the directory will not be deleted, nor will the macro receive an error.

Different file systems exhibit similar strange behavior in these cases.

Example

```
Sub Main
  On Error Goto ErrMake
  MkDir("test01")
  On Error Goto ErrRemove
  RmDir("test01")
ErrMake:
  MsgBox "The directory could not be created."
  Exit Sub
ErrRemove:
  MsgBox "The directory could not be removed."
  Exit Sub
End Sub
```

See Also Drive, Folder, and File Access on page 4

Rnd

Syntax `Rnd[(number)]`

Description Returns a random `Single` number between 0 and 1. If `number` is omitted, the next random number is returned. Otherwise, the `number` parameter has the following meaning:

If	Then
`number < 0`	Always returns the same number.
`Number = 0`	Returns the last number generated.
`Number > 0`	Returns the next random number.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  For x = -1 To 8
    y! = Rnd(1) * 100
    mesg = mesg & x & "  : " & y! & crlf
  Next x
  Session.Echo mesg & "Last form: " & Rnd
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

RSet

Syntax **`RSet destvariable = source`**

Description Copies the source string **`source`** into the destination string **`destvariable`**. If **`source`** is shorter in length than **`destvariable`**, then the string is right-aligned within **`destvariable`** and the remaining characters are padded with spaces. If **`source`** is longer in length than **`destvariable`**, then **`source`** is truncated, copying only the leftmost number of characters that will fit in **`destvariable`**. A runtime error is generated if **`source`** is **`Null`**.

The **`destvariable`** parameter specifies a string or variant variable. If **`destvariable`** is a variant containing empty, then no characters are copied. If **`destvariable`** is not convertible to a string, then a runtime error occurs. A runtime error results if **`destvariable`** is null.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Dim mesg,tmpstr$
  tmpstr$ = String$(40, "*")
  mesg = "Here are two strings that have been right-" & crlf
  mesg = mesg & "and left-justified in a 40-character string."
  mesg = mesg & crlf & crlf
  RSet tmpstr$ = "Right->"
  mesg = mesg & tmpstr$ & crlf
  LSet tmpstr$ = "<-Left"
  mesg = mesg & tmpstr$ & crlf
  Session.Echo mesg
End Sub
```

See Also Character and String Manipulation on page 3

RTrim, RTrim$

See Trim, Trim$, LTrim, LTrim$, RTrim, RTrim$; Character and String Manipulation on page 3.

S

SaveFilename$

Syntax `SaveFilename$[([title$ [,[extensions$] [helpfile,context]]])]`

Description Displays a dialog that prompts the user to select from a list of files and returns a **`String`** containing the full path of the selected file. The **`SaveFilename$`** function accepts the following parameters:

Parameter	Description
`title$`	String containing the title that appears on the dialog's caption. If this string is omitted, then **`"Save As"`** is used.
`extensions$`	String containing the available file types. If this string is omitted, then all files are used.
`helpfile`	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then **`context`** must also be specified.
`Context`	Number specifying the ID of the topic within **`helpfile`** for this dialog's help. If this parameter is specified, then **`helpfile`** must also be specified.

The SaveFilename$ function returns a full pathname of the file that the user selects. A zero-length string is returned if the user selects Cancel. If the file already exists, then the user is prompted to overwrite it.

If both the **`helpfile`** and **`context`** parameters are specified, then a Help button is added in addition to the OK and Cancel buttons. Context-sensitive help can be invoked by selecting this button or using the help key F1. Invoking help does not remove the dialog.

The **`extensions$`** parameter must be in the following format:

`description:ext[,ext][;description:ext[,ext]]...`

Placeholder	Description
description	Specifies the grouping of files for the user, such as All Files.
Ext	Specifies a valid file extension, such as *.BAT or *.?F?.

For example, the following are valid **extensions$** specifications:

```
"All Files:*"
"Documents:*.TXT,*.DOC"
"All Files:*;Documents:*.TXT,*.DOC"
```

Example

```
Sub Main
    e$ = "All Files:*.BMP,*.WMF;Bitmaps:*.BMP;Metafiles:*.WMF"
    f$ = SaveFilename$("Save Picture",e$)
    If Not f$ = "" Then
      MsgBox "User choose to save file as: " + f$
    Else
      MsgBox "User canceled."
    End If
End Sub
```

See Also Drive, Folder, and File Access on page 4; User Interaction on page 16

Second

Syntax **Second(*time*)**

Description Returns the second of the day encoded in the specified **time** parameter. The value returned is an **Integer** between 0 and 59 inclusive. The **time** parameter is any expression that converts to a **Date**.

Example

```
Sub Main
  xt# = TimeValue(Time$())
  xh# = Hour(xt#)
  xm# = Minute(xt#)
  xs# = Second(xt#)
  Session.Echo "The current time is: " & CStr(xh#) & ":" & CStr(xm#) _
    & ":" & CStr(xs#)
End Sub
```

See Also Time and Date Access on page 17

Seek (function)

Syntax **Seek(*filenumber*)**

Description Returns the position of the file pointer in a file relative to the beginning of the file. The **filenumber** parameter is a number that refers to an open file—the number passed to the **Open** statement. The value returned depends on the mode in which the file was opened:

File Mode	Returns
`Input`	Byte position for the next read
`Output`	Byte position for the next write
`Append`	Byte position for the next write
`Random`	Number of the next record to be written or read
`Binary`	Byte position for the next read or write

`The value returned is a Long` between 1 and 2147483647, where the first byte (or first record) in the file is 1.

Example

```
Sub Main
  Open "test.dat" For Random Access Write As #1
  For x = 1 To 10
    r% = x * 10
    Put #1,x,r%
  Next x
  y = Seek(1)
  Session.Echo "The current file position is: " & y
  Close
End Sub
```

See Also Drive, Folder, and File Access on page 4

Seek (statement)

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

Description Sets the position of the file pointer within a given file such that the next read or write operation will occur at the specified position. The **`Seek`** statement accepts the following parameters:

Parameter	Description
`filenumber`	Integer used to refer to the open file—the number passed to the **`Open`** statement.
`Position`	Long that specifies the location within the file at which to position the file pointer. The value must be between 1 and 2147483647, where the first byte (or record number) in the file is 1. For files opened in either Binary, Output, Input, or Append mode, **`position`** is the byte position within the file. For Random files, **`position`** is the record number.

A file can be extended by seeking beyond the end of the file and writing data there.

Example

```
Sub Main
  Open "test.dat" For Random Access Write As #1
  For x = 1 To 10
    rec$ = "Record#: " & x
    Put #1,x,rec$
  Next x
  Close
```

```
  Open "test.dat" For Random Access Read As #1
  Seek #1,9
  Get #1,,rec$
  Session.Echo "The ninth record = " & x
  Close
  Kill "test.dat"
End Sub
```

See Also Drive, Folder, and File Access on page 4

Select...Case

Syntax

```
Select Case testexpression
[Case expressionlist
  [statement_block]]
[Case expressionlist
  [statement_block]]
  .
  .
[Case Else
  [statement_block]]
End Select
```

Description Used to execute a block of statements depending on the value of a given expression. The **`Select Case`** statement has the following parts:

Part	Description
`testexpression`	Any numeric or string expression.
`Statement_block`	Any group of statements. If the **`testexpression`** matches any of the expressions contained in **`expressionlist`**, then this statement block will be executed.
`Expressionlist`	A comma-separated list of expressions to be compared against **`testexpression`** using any of the following syntax: **`expression [,expression]...expression To expression Is relational_operator expression`** The resultant type of expression in **`expressionlist`** must be the same as that of **`testexpression`**.

Multiple expression ranges can be used within a single **`Case`** clause. For example:

```
Case 1 to 10,12,15, Is > 40
```

Only the **`statement_block`** associated with the first matching expression will be executed. If no matching **`statement_block`** is found, then the statements following the **`Case Else`** will be executed.

A **`Select...End Select`** expression can also be represented with the **`If...Then`** expression. The use of the **`Select`** statement, however, may be more readable.

Example

```
'This example uses the Select...Case statement to return the
'type of key pressed.
Sub Main

Msgbox "Press any key.",ebOKOnly, "Select Case Example"
Session.KeyWait.Timeout = 10

Session.KeyWait.Start
KeyPress% = Session.KeyWait.Value

If Session.KeyWait.Status = smlWAITTIMEOUT Then
   MsgBox "Timeout period has expired."
Else
   Select Case KeyPress%
      Case 48 to 57
         TypeofKey$ = "number"
      Case 65 to 90, 97 to 122
         TypeofKey$ = "letter"
      Case Else
         TypeofKey$ = "non-alphanumeric"
   End Select
   MsgBox "The detected keystroke was a " & TypeofKey$ & "."
End If

End Sub
```

See Also Macro Control and Compilation on page 10

SelectBox

Syntax **SelectBox([*title*],*prompt*,*ArrayOfItems* [,*helpfile*,*context*])**

Description Displays a dialog that allows the user to select from a list of choices and returns an **Integer** containing the index of the item that was selected. The **SelectBox** statement accepts the following parameters:

Parameter	Description
title	Title of the dialog. This can be an expression convertible to a string. A runtime error is generated if **title** is null. If title is missing, then the default title is used.
prompt	Text to appear immediately above the listbox containing the items. This can be an expression convertible to a string. A runtime error is generated if **prompt** is null.
ArrayOfItems	Single-dimensioned array. Each item from the array will occupy a single entry in the listbox. A runtime error is generated if **ArrayOfItems** is not a single-dimensioned array. **ArrayOfItems** can specify an array of any fundamental data type (structures are not allowed). null and empty values are treated as zero-length strings.
Helpfile	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then **context** must also be specified.
Context	Number specifying the ID of the topic within **helpfile** for this dialog's help. If this parameter is specified, then **helpfile** must also be specified.

The value returned is an Integer representing the index of the item in the listbox that was selected, with the first item index to the lower bound of the array. If the lower bound of the array is 0 (the default), then the first item in the array is index 0, and a return value of -1 indicates that the user clicked Cancel. If the lower bound of the array is 1 (set with the Option Base statement), then the first item in the array is index 1, and a return value of 0 indicates that the user clicked Cancel.

Example

```
Sub Main
  Dim a$()
  AppList a$
  result% = SelectBox("Picker","Pick an application:",a$)
  If Not result% = -1 then
    Msgbox "User selected: " & a$(result%)
  Else
    Msgbox "User canceled"
  End If
End Sub
```

See Also Option Base on page 362; User Interaction on page 16

SendKeys

Syntax **SendKeys *string* [, [*wait*] [,*delay*]]**

Description Sends the specified keys to the active application, optionally waiting for the keys to be processed before continuing. If you're running the macro within the macro editor, **SendKeys** sends keystrokes to the editor. This statement is intended for use in applications; to send data to a host, use **Session.Send** instead.

The **SendKeys** statement accepts the following named parameters:

Parameter	Description
string	String containing the keys to be sent. The format for **string** is described below.
Wait	Boolean value. If True, then the compiler waits for the keys to be completely processed before continuing. The default value is False, which causes the compiler to continue macro execution while **SendKeys** finishes.
Delay	Integer specifying the number of milliseconds devoted for the output of the entire **string** parameter. It must be within the range 0 <= **delay** <= 32767. For example, if delay is 5000 (5 seconds) and the string parameter contains ten keys, then a key will be output every 1/2 second. If unspecified (0r 0), the keys will play back at full speed.

The **SendKeys** statement will wait for a prior **SendKeys** to complete before executing.

Specifying Keys

To specify any key on the keyboard, simply use that key, such as "a" for lowercase a, or "A" for uppercase a. Sequences of keys are specified by appending them together: **"abc"** or **"dir /w"**. Some keys have special meaning and are therefore specified in a special way—by enclosing them within braces. For example, to specify the percent sign, use "{%}". The following table shows the special keys:

Key	Special Meaning	Example	
+	Shift	**`"+{F1}"`**	Shift+F1
^	Ctrl	**`"^a"`**	Ctrl+A
~	Shortcut for Enter	**`"~"`**	Enter
%	Alt	**`"%F"`**	Alt+F
[]	No special meaning	**`"{[}"`**	Open bracket
{}	Used to enclose special keys	**`"{Up}"`**	Up arrow
()	Used to specify grouping	**`"^(ab)"`**	Ctrl+A, Ctrl+B

Keys that are not displayed when you press them are also specified within braces, such as **`{Enter}`** or **`{Up}`**. A list of these keys follows:

`{BkSp}`	**`{BS}`**	**`{Break}`**	**`{CapsLock}`**	**`{Clear}`**
`{Delete}`	**`{Del}`**	**`{Down}`**	**`{End}`**	**`{Enter}`**
`{Escape}`	**`{Esc}`**	**`{Help}`**	**`{Home}`**	**`{Insert}`**
`{Left}`	**`{NumLock}`**	**`{NumPad0}`**	**`{NumPad1}`**	**`{NumPad2}`**
`{NumPad3}`	**`{NumPad4}`**	**`{NumPad5}`**	**`{NumPad6}`**	**`{NumPad7}`**
`{NumPad8}`	**`{NumPad9}`**	**`{NumPad/}`**	**`{NumPad*}`**	**`{NumPad-}`**
`{NumPad+}`	**`{NumPad.}`**	**`{PgDn}`**	**`{PgUp}`**	**`{PrtSc}`**
`{Right}`	**`{Tab}`**	**`{Up}`**	**`{F1}`**	**`{Scroll Lock}`**
`{F2}`	**`{F3}`**	**`{F4}`**	**`{F5}`**	**`{F6}`**
`{F7}`	**`{F8}`**	**`{F9}`**	**`{F10}`**	**`{F11}`**
`{F12}`	**`{F13}`**	**`{F14}`**	**`{F15}`**	**`{F16}`**

Keys can be combined with Shift, Ctrl, and Alt using the reserved keys "+", "^", and "%" respectively:

For Key Combination	Use
Shift+Enter	**`"+{Enter}"`**
Ctrl+C	**`"^c"`**
Alt+F2	**`"%{F2}"`**

To specify a modifier key combined with a sequence of consecutive keys, group the key sequence within parentheses, as in the following example:

For Key Combination	Use
Shift+A, Shift+B	**"+(abc)"**
Ctrl+F1, Ctrl+F2	**"^({F1}{F2})"**

Use "~" as a shortcut for embedding **Enter** within a key sequence:

For Key Combination	Use
a, b, Enter, d, e	**"ab~de"**
Enter, Enter	**"~~"**

To embed quotation marks, use two quotation marks in a row:

For Key Combination	Use
"Hello"	**""Hello""**
a"b"c	**"a""b""c"**

Key sequences can be repeated using a repeat count within braces:

For Key Combination	Use
Ten **"a"** keys	**"{a 10}"**
Two Enter keys	**"{Enter 2}"**

Example

```
Sub Main
  id = Shell("Notepad.exe")
  AppActivate "Notepad"
  SendKeys "Hello, Notepad."    'Write some text.
  Sleep 2000
  SendKeys "%fs"          'Save file (simulate Alt+F, S keys).
  Sleep 2000
  SendKeys "name.txt{ENTER}"    'Enter name of new file to save.
  AppClose "Notepad"
End Sub
```

See Also Host Connections on page 7

Session (object)

The Session object gives you access to session-specific aspects of SmarTerm, including emulation settings and functions, host data access and capture, and basic host control.

Session.Application

Syntax **Session.Application**

Description Returns the session's application object.

Example

```
Dim App as Object
Set App = Session.Application
```

See Also Application and Session Features on page 11

Session.AutoWrap

VT, SCO, ANSI, and DG sessions only

Syntax `Session.AutoWrap`

Description Returns or sets the session’s autowrap state (boolean)

Example

```
Sub Main
  Dim AutoWrapState as Boolean
  AutoWrapState = Session.AutoWrap
  Session.AutoWrap = False
End Sub
```

See Also Application and Session Features on page 11

Session.Blink

Syntax `Session.Blink`

VT, SCO, ANSI, and DG sessions only

Description Returns or sets the blink attribute of the display presentation (boolean)

Example

```
Sub Main
  Dim BlinkState as Boolean
  BlinkState = Session.Blink
  Session.Blink = True
End Sub
```

See Also Application and Session Features on page 11

Session.Bold

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Bold`

Description Returns or sets the bold attribute of the display presentation (boolean).

Example

```
Sub Main
  Dim BoldState as Boolean
  BoldState = Session.Bold
  Session.Bold = False
End Sub
```

See Also Application and Session Features on page 11

Session.BufferFormatted

3270 and 5250 sessions only

Syntax `Session.BufferFormatted`

Description Returns **`True`** if the display buffer is formatted – if it contains any field definitions (boolean).

Value	Definition
True	Buffer is formatted
False	All other cases.

Example

```
Sub Main
  Dim BufForm as Boolean

  BufForm = Session.BufferFormatted
  If BufForm = True Then
     MsgBox "Buffer is formatted"
  End If
End Sub
```

See Also Application and Session Features on page 11

Session.BufferModified

3270 and 5250 sessions only

Syntax `Session.BufferModified`

Description Returns **`True`** if the display buffer has been modified (boolean). Possible values are:

Value	Description
True	Buffer has been modified (any MDT bits set)
False	All other cases.

Example

```
Sub Main
  Dim BufForm as Boolean

  BufForm = Session.BufferModified
  If BufMod = True Then
     MsgBox "Buffer has been modified"
  End If
End Sub
```

See Also Application and Session Features on page 11

Session.Caption

Syntax `Session.Caption`

Description Returns or sets SmarTerm's session window caption (string).

Example

```
Sub Main
  Dim CurrentCaption as String
  CurrentCaption = Session.Caption
  Session.Caption = "DG Session"
End Sub
```

See Also Application and Session Features on page 11

Session.Capture

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Capture(filename)`

where `filename` is the name of the file to write captured text (string).

Description Returns the completion status of the start-capture operation (boolean). Starts a capture operation, which writes incoming host data into the specified file.

Example

```
Sub Main
  Dim retval as Boolean
' Start a capture operation.
  Session.CaptureFileHandling = smlOVERWRITE
  retval = Session.Capture("FromHost.txt")
  If retval = FALSE Then
     Session.Echo "Error: Can't create file in Session.Capture"
     End
  End If
' Use LockStep to insure that the host and the PC stay in sync
  Dim LockSession as Object
  Set LockSession = Session.LockStep
  LockSession.Start
' Cause the host to start sending the desired information.
  Session.Send "TYPE REPORT1" + Chr$(13)
' Remain in capture mode until the ending string is detected from the host.
  Session.StringWait.MatchString = "End of Report"
  Session.StringWait.Start
' Terminate the capture.
  Session.EndCapture
' Cancel the LockStep state
  Set LockSession = Nothing
End Sub
```

See Also Drive, Folder, and File Access on page 4; Application and Session Features on page 11

Session.CaptureFileHandling

VT, SCO, ANSI, and DG sessions only

Syntax `Session.CaptureFileHandling`

Description Returns or sets the capture state (integer). Possible values are:

Value	Constant	Meaning
0	`smlOVERWRITE`	Overwrite an existing file.
1	`smlAPPEND`	Append to an existing file.
2	`smlPROMPTOVAPP`	Prompt whether to overwrite or append.

Example See the example for Session.Capture

See Also Drive, Folder, and File Access on page 4; Application and Session Features on page 11

Session.Circuit

Syntax `Session.Circuit`

Description Returns the `Circuit` object for the session. The `Session.Circuit` property is intended for use by external VBA controllers. The predefined `Circuit` object already exists for use by internal macros.

Example

```
Sub Main
Dim MyCircuit as Object
MyCircuit = Session.Circuit
End Sub
```

See Also Host Connections on page 7; Application and Session Features on page 11; Objects on page 18

Session.ClearScreen

Syntax `Session.ClearScreen`

Description Clears the SmarTerm screen. If the current session is text based (VT, ANSI, SCO, DG, or Wyse), it clears all text pages, resets graphic rendition and character attributes, resets all margins, performs a soft reset, and moves the cursor to the home position of the first page. If the current session is form-based (IBM 3270 or IBM 5250), the command clears all input fields.

Example

```
Sub Main
  Session.ClearScreen
End Sub
```

See Also Application and Session Features on page 11

Session.Close

Syntax `Session.Close`

Description Closes the SmarTerm session.

Example

```
Sub Main
  Dim nMsg as integer
  nMsg = Session.Echo ("Closing the current session. OK to proceed?", ebYesNo)
  If nMsg = ebYes Then
```

```
    Session.Close
  End If
End Sub
```

See Also Application and Session Features on page 11

Session.Collect (object)

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect`

Description Returns an object supporting access to SmarTerm's `Collect` feature. The **`Session.Collect`** object is used to extract data from the host-to-terminal data stream. There is one **`Collect`** object per-session. Its methods and properties can be divided into three categories: those used to initialize the wait object, those used to activate a wait, and those used to check the results of the wait. These categories are as follows:

Initialization

- Session.Collect.Reset
- Session.Collect.TermString
- Session.Collect.TermStringExact
- Session.Collect.Timeout
- Session.Collect.TimeoutMS
- Session.Collect.MaxCharacterCount
- Session.Collect.Consume

Activation

- Session.Collect.Start

Results

- Session.Collect.Status
- Session.Collect.CollectedCharacters
- Session.Collect.CollectedString

Note The **`Collect`** object automatically resets to its default (empty) state the first time any of its properties is set or any of its methods called after a previous **`Collect`** operation has completed.

In certain cases, it may be necessary to use the **`Lockstep`** feature to insure that the **`Collect`** object is presented with all data from the host that is significant. See the discussion of **`Session.Lockstep`** for further details.

Example

```
Sub Main
  Dim Report as String
  Session.Collect.TermString = "EndOfBlock"
  Session.Collect.Timeout = 100
  Session.Collect.Start
   If Session.Collect.Status = smlWAITSUCCESS Then
     MsgBox "CollectedCharacters: " &  _
             str$(Session.Collect.CollectedCharacters)
```

```
     MsgBox "Session.Collect.CollectedString: " &  _
             Session.Collect.CollectedString
  Else
    MsgBox "Timeout exceeded"
  End If
End Sub
```

See Also Character and String Manipulation on page 3; Application and Session Features on page 11; Objects on page 18

Session.Collect.CollectedCharacters

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect.CollectedCharacters`

Description Returns the number of characters in the collected string after a timeout condition or termination string match occurs (integer).

Example See the examples under Session.Collect (object).

See Also Character and String Manipulation on page 3; Application and Session Features on page 11

Session.Collect.CollectedString

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect.CollectedString`

Description Returns the collected string after a timeout condition or termination string match occurs (string).

Example See the examples under Session.Collect (object).

See Also Character and String Manipulation on page 3; Application and Session Features on page 11

Session.Collect.Consume

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect.Consume`

Description Returns or sets whether collected characters are presented to the display presentation (boolean). If this property is set `True`, the characters collected are not passed on to the display presentation.

Example See the examples under Session.Collect (object).

See Also Character and String Manipulation; Application and Session Features

Session.Collect.MaxCharacterCount

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect.MaxCharacterCount`

Description Returns or sets the maximum number of characters to collect before the collect operation terminates (integer).

Example See the examples under Session.Collect (object).

See Also Character and String Manipulation; Application and Session Features

Session.Collect.Reset

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect.Reset`

Description Resets the wait object's properties to their default values. The **`Collect`** object automatically resets to its default (empty) state when any of its properties is set or any of its methods is called after a previous **`Collect`** operation has completed.

Example See the examples under Session.Collect (object).

See Also Character and String Manipulation; Application and Session Features

Session.Collect.Start

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect.Start`

Description Returns a status value that indicates the reason that the wait ended (integer). This method activates the wait object, returning only when the specified conditions have been met. The status of the **`Collect`** operation is returned by the object's **`Start`** method and is also available through its **`Status`** property. The possible values are shown in the table below.

Value	Constant	Meaning
1	**`smlWAITSUCCESS`**	Successful match
-1	**`smlWAITTIMEOUT`**	Timeout
-2	**`smlWAITMAXCHARS`**	Maximum characters
-15	**`smlWAITERROR`**	Miscellaneous error

Example See the examples under Session.Collect (object).

See Also Character and String Manipulation; Application and Session Features

Session.Collect.Status

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect.Status`

Description Returns the most recent value returned by the `Start` method, or `0` if the wait object has been reset (integer). The status of the `Collect` operation is returned by the object's `Start` method and is also available through its `Status` property. The possible values are shown in the table below.

Value	Constant	Meaning
1	`smlWAITSUCCESS`	Successful match
-1	`smlWAITTIMEOUT`	Timeout
-2	`smlWAITMAXCHARS`	Maximum characters
-15	`smlWAITERROR`	Miscellaneous error

Example See the examples under Session.Collect (object).

See Also Character and String Manipulation; Application and Session Features

Session.Collect.TermString

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect.TermString`

Description Sets a pattern which, if detected in the host to terminal data stream during the course of a collect operation, terminates it. The comparison is case-insensitive. If case sensitivity is desired, set the `TermStringExact` property instead. This property overrides any previously established terminating pattern. If no terminating pattern is specified, no specific string terminates the collect operation.

Note This property is write-only.

Example See the examples under Session.Collect (object).

See Also Character and String Manipulation; Application and Session Features

Session.Collect.TermStringExact

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect.TermStringExact`

Description This property sets a pattern which, if detected in the host to terminal data stream during the course of a collect operation, terminates it. The comparison is case-sensitive. If case sensitivity is not desired, set the `TermString` property instead. This property overrides any previously established terminating pattern. If no terminating pattern is specified, no specific string terminates the collect operation.

Note This property is write-only.

Example See the examples under Session.Collect (object).

See Also Character and String Manipulation; Application and Session Features

Session.Collect.Timeout

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect.Timeout`

Description Returns or sets the maximum number of seconds allowed for the collect operation (integer).

Example See the examples under Session.Collect (object).

See Also Character and String Manipulation; Application and Session Features

Session.Collect.TimeoutMS

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Collect.TimeoutMS`

Description Sets the maximum number of milliseconds to allow for the collect operation (integer).

Note This property is write-only.

Example See the examples under Session.Collect (object).

See Also Character and String Manipulation; Application and Session Features

Session.Column

Syntax `Session.Column`

Description Returns or sets where the cursor is placed in the current SmarTerm session window.

Example

```
Sub Main
  Dim CurrentCol as Integer
  CurrentCol = Session.Column
  Session.Column = CurrentCol + 10
End Sub
```

See Also Application and Session Features

Session.Concealed

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Concealed`

Description Returns or sets the concealed attribute of the display presentation (boolean).

Example

```
Sub Main
  Dim ConcealedState as Boolean
  ConcealedState = Session.Concealed
  Session.Concealed = True
End Sub
```

See Also Application and Session Features

Session.ConfigInfo

Syntax **`Session.ConfigInfo (infotype)`**

Description Returns the requested SmarTerm information (string). **`infotype`** specifies the type of information to return (integer). The possible values are:

Value	Constant	Meaning
0	**`smlSESSIONPATH`**	Full path of the SmarTerm session (STW) file
2	**`smlINSTALLPATH`**	Full path to where SmarTerm is installed

Example

```
Sub Main
  Dim StwPath as String
  Dim InstPath as string
  StwPath = Session.ConfigInfo(smlSESSIONPATH)
  Session.Echo "SmarTerm session file is " & StwPath
  InstPath = Session.ConfigInfo(smlINSTALLPATH)
  Session.Echo "SmarTerm installation directory is " & InstPath
End Sub
```

See Also Application and Session Features

Session.Connected

Syntax **`Session.Connected`**

Description Returns a boolean representing the session's connection status. If **`True`**, a connection is established.

Example

```
Sub Main
  Dim fConnected as Boolean
  fConnected = Session.Connected
  If fConnected Then
     Session.Echo "You are connected."
  End If
End Sub
```

See Also Host Connections; Application and Session Features

Session.DialogView

3270 and 5250 sessions only

Syntax `Session.DialogView`

Description Returns or sets the session's DialogView state (Boolean), allowing you to toggle the DialogView feature on or off.

Example

```
Sub Main
' This example displays the current DialogView state
' and then toggles it.

  Dim fIsDialogView as Boolean
  Dim strDialogView as String

' Get the current state of DialogView and inform user
  fIsDialogView = Session.DialogView
  If fIsDialogView = TRUE then
    strDialogView = "The emulator is in DialogView mode"
  Else
    strDialogView = "The emulator is in Emulation mode"
  End If

' Now switch modes
  MsgBox strDialogView + " Switching modes..."
  Session.DialogView = Not fIsDialogView
End Sub
```

See Also User Interaction; Application and Session Features

Session.DoMenuFunction

Syntax `Session.DoMenuFunction menuitem$`

where `menuitem$` is the menu item to trigger (string).

Note The list presented here is complete; the availability of the actual values varies depending on the capability of the current session type.

Description Triggers a session-based menu action in SmarTerm. Possible values:

ConnectionClearPort	FilePrint	ToolsFTPDragAndDrop
ConnectionConnect	FileSaveSession	ToolsHotSpots
ConnectionDisconnect	FileSaveSessionAs	ToolsKeyboardMaps
ConnectionOnline	FileSendMail	ToolsMacro
ConnectionProperties	PrinterCancel	ToolsReceiveFile
ConnectionSendBreak	PrinterFlush	ToolsReplayCapturedFile
ConnectionStartTrace	PrinterPA1	ToolsSendFile
EditClearHistory	PrinterPA2	ToolsSmarTermButtons
EditClearScreen	PrinterTest	ToolsSmartMouse

`EditCopy`	`PropertiesEmulation`	`ToolsStartCapture`
`EditCopyScreenToHistory`	`PropertiesFileTransferProperties`	`ToolsStopCapture`
`EditCopyTable`	`PropertiesFileTransferProtocol`	`ToolsTriggers`
`EditCopyToFile`	`PropertiesHardReset`	`ViewDialogView`
`EditPaste`	`PropertiesResetTerminal`	`ViewHotSpots`
`EditPasteFromFile`	`PropertiesSessionOptions`	`ViewTerminal`
`EditSelectScreen`	`PropertiesSoftReset`	`ViewTriggers`
`EditSelectScreenAndHistory`	`ToolsFTPCommandMode`	`ViewSmarTermButtons`
`FileClose`		

Example

```
Sub Main
  Session.DoMenuFunction "ToolsMacros"
End Sub
```

See Also Application and Session Features

Session.Echo

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Echo text$`

where **`text$`** is the text to display (string).

Description Displays text in the window as if it had been sent by the host.

Example

```
Sub Main
  Session.Echo ""About to connect to host"
  Session.Echo "Please be ready to log in<CR><LF>"
End Sub
```

See Also Application and Session Features; User Interaction

Session.EmulationInfo

Syntax `Session.EmulationInfo(infotype)`

where **`infotype`** specifies the information to return (integer).

Description Returns either the emulation family or the emulation level (string). Possible values are:

Value	Constant	Meaning
0	`smlEMUFAMILY`	The emulation family.
1	`smlEMULEVEL`	The emulation level.

Note Calling **`Session.EmulationInfo(smlEMUFAMILY)`** will return the string **`"NVT"`** if the actual terminal type is yet to be established.

Example

```
Sub Main
  Dim EmulationFamily as String
  Dim EmulationLevel as String
  EmulationFamily = Session.EmulationInfo(smlEMUFAMILY)
  Session.Echo "Your current session type is " & EmulationFamily
  EmulationLevel = Session.EmulationInfo(smlEMULEVEL)
  Session.Echo "Your current operating level is " & EmulationLevel
End Sub
```

See Also Application and Session Features

Session.EndCapture

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.EndCapture`**

Description Stops a capture operation.

Example See the example for Session.Capture.

See Also Drive, Folder, and File Access; Application and Session Features

Session.EventWait (object)

3270 and 5250 sessions only

Syntax **`Session.EventWait`**

Description Returns an object supporting access to SmarTerm's **`EventWait`** feature. The **`Session.EventWait`** object is used to pause macro execution pending the receipt or issue of certain events. There is one **`EventWait`** object per-session. Its methods and properties can be divided into three categories: those used to initialize the wait object, those used to activate a wait, and those used to check the results of the wait. These categories are as follows:

Initialization

- Session.EventWait.EventType
- Session.EventWait.MaxEventCount
- Session.EventWait.Reset
- Session.EventWait.Timeout
- Session.EventWait.TimeoutMS

Activation

- Session.EventWait.Start

Results

- Session.EventWait.EventCount
- Session.EventWait.Status

The **`EventWait`** object automatically resets to its default (empty) state the first time any of its properties is set or any of its methods called after a previous **`EventWait`** operation has completed.

In certain cases, it may be necessary to use the **Lockstep** feature to insure that the **EventWait** object is presented with all data from the host that is significant. See the discussion of **Session.Lockstep** for further details.

Example

```
Sub Main
  ' Wait for a PAGERECEIVED event
  Session.Eventwait.EventType = smlPAGERECEIVED
  Session.Eventwait.Start
  ' Wait for a PAGESENT event
  Session.Eventwait.EventType = smlPAGESENT
  Session.Eventwait.Start
  ' Wait for 3 PAGERECEIVED events, or 30 seconds,
  ' whichever comes first.
  Session.Eventwait.EventType = smlPAGERECEIVED
  Session.EventWait.MaxEventCount = 3
  Session.EventWait.Timeout = 30
  Session.Eventwait.Start
  If Session.EventWait.Status = smlWAITTIMEOUT Then
     MsgBox "Timeout exceeded, Total events detected: " & _
        str$(Session.EventWait.EventCount)
  End If
End Sub
```

See Also Host Connections on page 7; Application and Session Features on page 11; Objects on page 18

Session.EventWait.EventCount

3270 and 5250 sessions only

Syntax **Session.EventWait.EventCount**

Description Returns the number of events that occurred during the wait period (integer).

Example See the examples under Session.EventWait (object).

See Also Host Connections on page 7; Application and Session Features on page 11

Session.EventWait.EventType

3270 and 5250 sessions only

Syntax **Session.EventWait.EventType**

Description Returns or sets the type of event to wait for (integer). The possible values are:

Value	Constant	Meaning
1	**smlPAGERECEIVED**	A form has been received from the host.
2	**smlPAGESENT**	A form has been sent to the host.

Example See the examples under Session.EventWait (object).

See Also Host Connections on page 7; Application and Session Features on page 11

Session.EventWait.MaxeventCount

3270 and 5250 sessions only

Syntax `Session.EventWait.MaxEventCount`

Description Returns or sets the maximum number of events to allow to pass while a wait is active (integer).

Example See the examples under Session.EventWait (object).

See Also Host Connections on page 7; Application and Session Features on page 11

Session.EventWait.Reset

3270 and 5250 sessions only

Syntax `Session.EventWait.Reset`

Description Resets the wait object's properties to their default values. The **`EventWait`** object automatically resets to its default (empty) state when any of its properties is set or any of its methods called after a previous **`EventWait`** operation has completed.

Example See the examples under Session.EventWait (object).

See Also Host Connections on page 7; Application and Session Features on page 11

Session.EventWait.Start

3270 and 5250 sessions only

Syntax `Session.EventWait.Start`

Description Returns a status value that indicates the reason that the wait ended (integer). Activates the wait object, returning only when the specified conditions have been met. The status of the EventWait operation is returned by the object's **`Start`** method and is also available through its **`Status`** property. The possible values are shown in the table below.

Value	Constant	Meaning
1	**`smlWAITSUCCESS`**	Successful match
-1	**`smlWAITTIMEOUT`**	Timeout
-2	**`smlWAITMAXEVENTS`**	Maximum events
-15	**`smlWAITERROR`**	Miscellaneous error

Example See the examples under Session.EventWait (object).

See Also Host Connections on page 7; Application and Session Features on page 11

Session.EventWait.Status

3270 and 5250 sessions only

Syntax `Session.EventWait.Status`

Description Returns the most recent value returned by the `Start` method, or 0 if the wait object has been reset (integer). The status of the `EventWait` operation is returned by the object's `Start` method and is also available through its `Status` property. The possible values are shown in the table below.

Value	Constant	Meaning
1	`smlWAITSUCCESS`	Successful match
-1	`smlWAITTIMEOUT`	Timeout
-2	`smlWAITMAXEVENTS`	Maximum events
-15	`smlWAITERROR`	Miscellaneous error

Example See the examples under Session.EventWait (object).

See Also Host Connections on page 7; Application and Session Features on page 11

Session.EventWait.Timeout

3270 and 5250 sessions only

Syntax `Session.EventWait.Timeout`

Description Returns or sets the wait object's timeout value, in seconds (integer).

Example See the examples under Session.EventWait (object).

See Also Host Connections on page 7; Application and Session Features on page 11

Session.EventWait.TimeoutMS

3270 and 5250 sessions only

Syntax `Session.EventWait.TimeoutMS`

Description Sets the wait object's timeout value, in milliseconds (integer).

Example See the examples under Session.EventWait (object).

See Also Host Connections on page 7; Application and Session Features on page 11

Session.FieldEndCol

3270 and 5250 sessions only

Syntax `Session.FieldEndCol`

Description Returns the ending column number (1 based) of the field where the cursor resides. On an unformatted display, this property always defaults to the number of columns on the display page.

Note This property is read-only.

Example

```
Sub Main
  Dim StartRow as Integer
  Dim StartCol as Integer
  Dim EndRow as Integer
  Dim EndCol as Integer
  Dim CurScn as String
  StartRow = Session.FieldStartRow
  StartCol = Session.FieldStartCol
  EndRow   = Session.FieldEndRow
  EndCol   = Session.FieldEndCol
  CurScn = Session.NativeScreenText(StartRow,StartCol,EndRow,EndCol)
  MsgBox "The entire current field where the cursor is placed " &_
         "is (EBCDIC)" & CurScn
End Sub
```

See Also Application and Session Features on page 11

Session.FieldEndRow

3270 and 5250 sessions only

Syntax `Session.FieldEndRow`

Description Returns the ending row number (1 based) of the field where the cursor resides. On an unformatted display, this property always defaults to the number of lines on the display page.

Note This property is read-only.

Example

```
Sub Main
  Dim StartRow as Integer
  Dim StartCol as Integer
  Dim EndRow as Integer
  Dim EndCol as Integer
  Dim CurScn as String
  StartRow = Session.FieldStartRow
  StartCol = Session.FieldStartCol
  EndRow   = Session.FieldEndRow
  EndCol   = Session.FieldEndCol
  CurScn = Session.NativeScreenText(StartRow,StartCol,EndRow,EndCol)
  MsgBox "The entire current field where the cursor is placed " &_
         "is (EBCDIC)" & CurScn
End Sub
```

See Also Application and Session Features on page 11

Session.FieldModified

5250 sessions only

Syntax `Session.FieldModified`

Description Returns whether the current field (the field that the cursor is in) has been modified (boolean). `Session.FieldModified` returns one of the following values:

Value	Definition
True	The field in which the cursor resides has been modified.
False	Buffer is not formatted or field is not modified.

Example

```
Sub Main
Dim fModified as Boolean
fModified = Session.FieldModified
If fModified Then
  MsgBox "Field is modified."
End If
```

Session.FieldStartCol

3270 and 5250 sessions only

Syntax `Session.FieldStartCol`

Description Returns the beginning column number (1 based) of the field where the cursor resides (integer). On an unformatted display, this property always has the value of 1. This property is read-only.

Example

```
Sub Main
  Dim StartRow as Integer
  Dim StartCol as Integer
  Dim EndRow as Integer
  Dim EndCol as Integer
  Dim CurScn as String
  StartRow = Session.FieldStartRow
  StartCol = Session.FieldStartCol
  EndRow   = Session.FieldEndRow
  EndCol   = Session.FieldEndCol
  CurScn = Session.NativeScreenText(StartRow,StartCol,EndRow,EndCol)
  MsgBox "The entire current field where the cursor is placed " &_
         "is (EBCDIC)" & CurScn
End Sub
```

See Also Application and Session Features on page 11

Session.FieldStartRow

3270 and 5250 sessions only

Syntax `Session.FieldStartRow`

Description Returns the beginning row number (1 based) of the field where the cursor resides (integer). On an unformatted display, this property always has the value of 1. This property is read-only.

Example

```
Sub Main
  Dim StartRow as Integer
  Dim StartCol as Integer
  Dim EndRow as Integer
  Dim EndCol as Integer
  Dim CurScn as String
  StartRow = Session.FieldStartRow
  StartCol = Session.FieldStartCol
  EndRow   = Session.FieldEndRow
  EndCol   = Session.FieldEndCol
  CurScn = Session.NativeScreenText(StartRow,StartCol,EndRow,EndCol)
  MsgBox "The entire current field where the cursor is placed " &
         "is (EBCDIC)" & CurScn
End Sub
```

See Also Application and Session Features on page 11

Session.FieldText

3270 and 5250 sessions only

Syntax `Session.FieldText (row, col)`

Description Returns the text (in ASCII/ISO-Latin1) from the field containing the specified cursor position. If the field is numeric, the property returns the text representation of the numbers, including a plus or minus sign for positive and negative numbers. If the text cannot be returned for some reason, the property returns an empty string.

Note 5250 hosts respond to this property only if the specified location has been defined as an unprotected field. Unlike 3270 host applications, screen locations on 5250 hosts are not automatically defined as fields, but *must* be defined by the application.

Parameters are:

Parameter	Definition
`row`	The row containing the desired text (integer).
`col`	The column containing the desired text (integer).

If the row or column value is less than or equal to 0, the function defaults to the current cursor row or column, respectively. A row or column value outside the range is truncated to fit within the display.

Note This property is read-only.

Example

```
Sub Main
  Dim FieldData as String
  FieldData = Session.FieldText(Session.Row, Session.Column)
  MsgBox "Current field displays " & FieldData
End Sub
```

Session.FontAutoSize

Syntax `Session.FontAutoSize`

Description Returns or sets the auto-font-size state of characters displayed in the current SmarTerm session (boolean). When set `True`, the font size is set automatically based on the window size.

Example

```
Sub Main
  Dim AutoFontState as Boolean
  AutoFontState = Session.FontAutoSize
  Session.FontAutoSize = True
End Sub
```

See Also Application and Session Features on page 11

Session.FontHeight

Syntax `Session.FontHeight`

Description Returns or sets the font height of characters displayed in the current SmarTerm session (integer).

Example

```
Sub Main
  Dim Height as Integer
  Height = Session.FontHeight
  Session.FontHeight = 2 * Height
End Sub
```

See Also Application and Session Features on page 11

Session.FontWidth

Syntax `Session.FontWidth`

Description Returns or sets the font width of characters displayed in the current SmarTerm session (integer).

Example

```
Sub Main
  Dim Width as Integer
  Width = Session.FontWidth
  Session.FontWidth = 2 * Width
End Sub
```

See Also Application and Session Features on page 11

Session.GetMostRecentTriggerName

Syntax `Session.GetMostRecentTriggerName`

Description Returns a string containing the name of the most recently fired trigger. Note that this property is not cleared when the host clears the matching pattern (retrieved with Session.GetMostRecentTriggerPattern) from the screen.

Example

```
Sub Main

Dim TriggerName$
TriggerName$ = Session.GetMostRecentTriggerName

If TriggerName$ = "Start Page" Then
   MsgBox "We are on the starting page of the host screen."
End If
End Sub
```

See Also Application and Session Features on page 11

Session.GetMostRecentTriggerPattern

Syntax **`Session.GetMostRecentPattern`**

Description Returns a string containing the the most recently match trigger pattern. Note that this property is not cleared when the host clears the matching pattern from the screen.

Example

```
Sub Main

Dim TriggerPattern$
TriggerPattern$ = Session.GetMostRecentTriggerPattern

If TriggerPattern$ = "AS/400 Main Menu" Then
   MsgBox "We are on the starting page of the host screen."
End If
End Sub
```

See Also Application and Session Features on page 11

Session.HotSpotsActive

Syntax **`Session.HotSpotsActive [= TRUE | FALSE]`**

Description Returns or sets whether the current HotSpots file is visible or not (Boolean).

Example

```
'This example toggles the current HotSpots file.
Sub Main
  CurrentFile$ = Session.HotSpotsFileName

' First, see if there's a file to toggle.
  If CurrentFile$ = "" Then
    MsgBox "No HotSpots loaded."

' Now turn it on if it's off, off if it's on.
  Else
    If Session.HotSpotsActive = True Then
      Session.HotSpotsActive = False
      MsgBox "HotSpots " & CurrentFile$ & " now OFF."
    Else
```

```
      Session.HotSpotsActive = True
      MsgBox "HotSpots " & CurrentFile$ & " now ON."
    End If
  End If
End Sub
```

See Also Application and Session Features on page 11; User Interaction on page 16

Session.HotSpotsFileName

Syntax `Session.HotSpotsFileName [= Filename]`

Description Returns the name of the current HotSpots file (string). If you specify a HotSpots file with the Filename parameter (string), then the program attempts to load that file. This usage is therefore similar to the Session.SetHotSpotsFile method, except that there is no built-in error-checking.

`Filename` can specify the complete path to the desired HotSpots file. If no path is specified, the program looks in the User HotSpot folder.

Example

```
'This example reports the name of the current HotSpots file.
' If no file is loaded, it loads DEFAULT.HOT
Sub Main
  CurrentFile$ = Session.HotSpotsFileName

  If CurrentFile$ <> "" Then
    MsgBox "Current HotSpots file: ." & CurrentFile$
  Else
    If (Session.HotSpotsFileName = "DEFAULT.HOT")= TRUE Then
      MsgBox "HotSpots DEFAULT.HOT now loaded."
    Else
      MsgBox "No HotSpots available."
    End If
  End If
End Sub
```

See Also Application and Session Features on page 11; User Interaction on page 16

Session.InitialMouseCol

Syntax `Session.InitialMouseCol`

Description Returns the mouse's column position at the time a macro was started (integer).

`Session.InitialMouseCol` and `Session.InitialMouseRow` contain the text column and row (respectively) that the mouse pointer was over when the script was started. If the mouse pointer is outside of the configuration window, the values are clipped to within the window.

The value within this property is only meaningful when accessed from an internal macro. When accessed through an external OLE Automation controller, the value returned will be the one established when the last internal macro was executed.

Example

```
Sub Main
  Dim StartX as Integer
  Dim StartY as Integer
  StartX = Session.InitialMouseCol
  StartY = Session.InitialMouseRow
  Msgbox "Initial mouse position was Row: " & str(StartY) & " Col: " & str(StartX)
End Sub
```

See Also Application and Session Features on page 11

Session.InitialMouseRow

Syntax `Session.InitialMouseRow`

Description Returns the mouse's row position at the time a macro was started (integer).

`Session.InitialMouseCol` and `Session.InitialMouseRow` contain the text column and row (respectively) that the mouse pointer was over when the script was started. If the mouse pointer is outside of the configuration window, the values are clipped to within the window.

The value within this property is only meaningful when accessed from an internal macro. When accessed through an external OLE Automation controller, the value returned will be the one established when the last internal macro was executed.

Example

```
Sub Main
  Dim StartX as Integer
  Dim StartY as Integer
  StartX = Session.InitialMouseCol
  StartY = Session.InitialMouseRow
  Msgbox "Initial mouse position was Row: " & str(StartY) & " Col: " & str(StartX)
End Sub
```

See Also Application and Session Features on page 11

Session.InsertMode

3270 and 5250 sessions only

Syntax `Session.InsertMode`

Description Returns `True` if the terminal is currently in insert mode (Boolean).

Example

```
Sub Main
  Dim InsertMode as Boolean
  InsertMode = Session.InsertMode
  If InsertMode = TRUE Then
     MsgBox "You are in insert mode."
  End If
End Sub
```

Session.InterpretControls

VT, SCO, ANSI, and DG sessions only

Syntax `Session.InterpretControls`

Description Returns or sets whether control characters are interpreted or displayed in the current SmarTerm session (boolean)

Example

```
Sub Main
  Dim ControlState as Boolean
  ControlState = Session.InterpretControls
  Session. InterpretControls = True
End Sub
```

Session.Inverse

VT, SCO, ANSI, and DG sessions only

Syntax `Session.Inverse`

Description Returns or sets the inverse attribute of the current session's display presentation (boolean).

Example

```
Sub Main
  Dim Inverse State as Boolean
  InverseState = Session.Inverse
  Session.Inverse = True
End Sub
```

See Also Application and Session Features on page 11)

Session.IsFieldMark

3270 sessions only

Syntax `Session.IsFieldMark(row, col)`

Description Returns **`True`** if the cursor position containing the specified row and column is the beginning of a field (a field mark); returns **`False`** in all other cases (boolean). Parameters are:

Parameter	Description
`row`	The row to test (integer).
`col`	The column to test (integer)

Example

```
Sub Main
  Dim Fieldmark as Boolean

  Fieldmark = Session.IsFieldMark(4,11)
  If Fieldmark = True Then
     MsgBox "You are at the beginning of a field"
  End If
End Sub
```

Session.IsNumeric

3270 and 5250 sessions only

Syntax `Session.IsNumeric(row, col)`

Description Returns `True` if the specified character position is within a numeric field (boolean). Parameters are:

Parameter	Description
`row`	The row to test (integer).
`col`	The column to test (integer)

Example

```
Sub Main
  Dim IsNum as Boolean

  IsNum = Session.IsNumeric(Session.Row, Session.Column)
  If IsNum = True Then
     MsgBox "Cursor is in a numeric field"
  End If
End Sub
```

Session.IsProtected

3270 and 5250 sessions only

Syntax `Session.IsProtected(row, col)`

Description Returns an indication of whether the specified character position is within a protected field (integer). Parameters are:

Parameter	Description
`row`	The row to test (integer).
`col`	The column to test (integer)

Returns 0 if the specified cursor position is in an unprotected field; returns -1 if the position is a field mark or an unprotected field; returns 1 in all other cases. If row or col is less than or equal to 0, the function defaults to the current cursor row or column, respectively. A row or column outside the range is truncated to fit within the display.

Example

```
Sub Main
  Dim IsProtected as Integer
' Is there a protected field at row 11, column 4?
  IsProtected = Session.IsProtected(11, 4)
  If IsProtected = 1 Then
     MsgBox "Row 11, Column 4 is a protected field"
  End If
End Sub
```

Session.KeyboardLocked

3270 and 5250 sessions only

Syntax **`Session.KeyboardLocked`**

Description Returns the state of the keyboard in SmarTerm (integer). Evaluates to 0 if the keyboard is unlocked; it evaluates to non-zero for lock conditions. If the lock was the result of an error (alphabetic character in a numeric field, protected field, field overflow, or “Prog” error), the value is less than 0. If the lock is the result of a system command or function key, the value is greater than 0.

Example

```
Sub Main
  Dim KeyboardLocked as Integer
  Dim UserMessage as string
  KeyboardLocked = Session.KeyboardLocked
  if KeyboardLocked = 0 Then
     UserMessage = "Keyboard is unlocked."
  Elseif KeyboardLocked > 0 Then
       UserMessage = "Keyboard locked from a command or key."
  Else
       UserMessage = "Keyboard locked from field overflow."
  End If
  MsgBox UserMessage

End Sub
```

Session.KeyWait (object)

Syntax **`Session.KeyWait`**

Description Returns an object supporting access to SmarTerm’s **`KeyWait`** feature.

The **`Session.KeyWait`** object is used to wait for specific keystrokes or mouse button clicks to be entered. There is one **`KeyWait`** object per-session. Its methods and properties can be divided into three categories: those used to initialize the wait object, those used to activate a wait, and those used to check the results of the wait. These categories are as follows:

Initialization

Session.KeyWait.KeyCode
Session.KeyWait.KeyType
Session.KeyWait.Timeout
Session.KeyWait.TimeoutMS
Session.KeyWait.MaxKeyCount
Session.KeyWait.Reset

Activation

Session.KeyWait.Start

Results

Session.KeyWait.Status
Session.KeyWait.Value
Session.KeyWait.KeyCount

The **KeyWait** object automatically resets to its default (empty) state the first time any of its properties is set or any of its methods called after a previous **KeyWait** operation has completed.

Example

```
Sub Main
  ' Wait for any key, using the Reset method to insure the following defaults:
  '     KeyType = smlKEYWCOUNT
  '     MaxKeyCount = 0
  Session.KeyWait.Reset
  Session.KeyWait.Start
  ' Wait for any key, but give up after 5 seconds
  Session.KeyWait.Timeout = 5
  Session.KeyWait.Start
  If Session.KeyWait.Status = smlWAITTIMEOUT Then
     Session.Echo "Tired of waiting"
  Else
     Session.Echo "Detected keystroke: " & str$(Session.Keywait.Value)
  End If
  ' Wait for either an 'a' or an 'A'
  Session.KeyWait.KeyCode = asc("A")
  Session.KeyWait.KeyType = smlKEYWNONEXACT
  Session.KeyWait.Start
' Wait for an 'A'
  Session.KeyWait.KeyCode = asc("A")
  Session.KeyWait.KeyType = smlKEYWEXACT
  Session.KeyWait.Start
  ' Wait for three keystrokes
  Session.KeyWait.KeyType = smlKEYWCOUNT
  Session.KeyWait.MaxKeyCount = 3
  Session.KeyWait.Start
  ' Wait for scancode 33  (the 'F' key on US keyboards)
  Session.KeyWait.KeyCode = 33
  Session.KeyWait.KeyType = smlKEYWSCAN
  Session.KeyWait.Start
  ' Wait for DEC key 101
  Session.KeyWait.KeyCode = 101
  Session.KeyWait.KeyType = smlKEYWDECKEY
  Session.KeyWait.Start
  ' Wait for virtual key 69
  Session.KeyWait.KeyCode = 69
  Session.KeyWait.KeyType = smlKEYWVIRTUAL
  Session.KeyWait.Start
  ' Wait for the click of a mouse button
  Session.KeyWait.KeyType = smlKEYWBUTTON
  Session.KeyWait.Start
  Select Case Session.KeyWait.Value
    Case 1
        Session.Echo "Detected left mouse button"
    Case 2
        Session.Echo "Detected middle mouse button"
    Case 3
        Session.Echo "Detected right mouse button"
  End Select
End Sub
```

See Also Host Connections on page 7; Application and Session Features on page 11; Objects on page 18

Session.KeyWait.KeyCode

Syntax `Session.KeyWait.KeyCode`

Description Returns or sets the **`KeyCode`** value to wait for (integer).

Note Be sure to also set the **`KeyType`** property to qualify the **`KeyCode`** value.

Example See the examples under Session.KeyWait (object).

See Also Application and Session Features on page 11

Session.KeyWait.KeyCount

Syntax `Session.KeyWait.KeyCount`

Description Returns the number of keys detected by the wait object before a return was made from the **`Start`** method (integer).

Example See the examples under Session.KeyWait (object).

See Also Application and Session Features on page 11

Session.KeyWait.KeyType

Syntax `Session.KeyWait.KeyType`

Description Returns or sets the type of key to wait for (integer). This property qualifies the value set within the **`KeyCode`** property. The possible values are:

Value	Constant	Meaning
1	**`smlKEYWEXACT`**	Non-case folded character/ASCII code
2	**`smlKEYWNONEXACT`**	Non-case folded character/ASCII code
3	**`smlKEYWSCAN`**	PC scan code
4	**`smlKEYWVIRTUAL`**	Virtual key code (Windows specific)
5	**`smlKEYWDECKEY`**	Emulation specific key code (DECKEY in PSL)
6	**`smlKEYWBUTTON`**	Mouse button
7	**`smlKEYWCOUNT`**	Any key (Use the count)

Example See the examples under Session.KeyWait (object).

See Also Application and Session Features on page 11

Session.KeyWait.MaxKeyCount

Syntax `Session.KeyWait.MaxKeyCount`

Description Returns or sets the maximum number of keys to wait for before returning from the **`Start`** method (integer).

Example See the examples under Session.KeyWait (object).

See Also Application and Session Features on page 11

Session.KeyWait.Reset

Syntax **`Session.KeyWait.Reset`**

Description Resets the wait object's properties to their default values. The **`KeyWait`** object automatically resets to its default (empty) state when any of its properties is set or any of its methods called after a previous **`KeyWait`** operation has completed.

Example See the examples under Session.KeyWait (object).

See Also Application and Session Features on page 11

Session.KeyWait.Start

Syntax **`Session.KeyWait.Start`**

Description Returns a status value that indicates the reason that the wait ended (integer). Activates the wait object, returning only when the specified conditions have been met. The status of the **`KeyWait`** operation is returned by the object's **`Start`** method and is also available through its **`Status`** property. The possible values are shown in the table below.

Value	Constant	Meaning
1	**`smlWAITSUCCESS`**	Successful match
-1	**`smlWAITTIMEOUT`**	Timeout
-2	**`smlWAITMAXCHARS`**	Maximum characters
-15	**`smlWAITERROR`**	Miscellaneous error

Example See the examples under Session.KeyWait (object).

See Also Application and Session Features on page 11

Session.KeyWait.Status

Syntax **`Session.KeyWait.Status`**

Description Returns the most recent value returned by the **`Start`** method, or 0 if the wait object has been reset (integer). The status of the **`KeyWait`** operation is returned by the object's **`Start`** method and is also available through its **`Status`** property. The possible values are shown in the table below.

Value	Constant	Meaning
1	`smlWAITSUCCESS`	Successful match
-1	`smlWAITTIMEOUT`	Timeout
-2	`smlWAITMAXCHARS`	Maximum characters
-15	`smlWAITERROR`	Miscellaneous error

Example See the examples under Session.KeyWait (object).

See Also Application and Session Features on page 11

Session.KeyWait.Timeout

Syntax `Session.KeyWait.Timeout`

Description Returns or sets the wait object's timeout value, in seconds (integer). The default value is 0, which means that no timeout will occur.

Example See the examples under Session.KeyWait (object).

See Also Application and Session Features on page 11

Session.KeyWait.TimeoutMS

Syntax `Session.KeyWait.TimeoutMS`

Description Sets the wait object's timeout value, in milliseconds (integer). The default value is `0`, which means that no timeout will occur.

Example See the examples under Session.KeyWait (object).

See Also Application and Session Features on page 11

Session.KeyWait.Value

Syntax `Session.KeyWait.Value`

Description Returns the keystroke value that caused the `Start` method to return (integer).

Example See the examples under Session.KeyWait (object).

See Also Application and Session Features on page 11

Session.Language

Syntax `Session.Language`

Description Returns or sets a language for the session (integer). Possible values are:

Value	Constant	Meaning
1031	**smlGERMAN**	German.
1033	**smlENGLISH**	English.
1036	**smlFRENCH**	French.
1034	**smlSPANISH**	Spanish.

See Also **Application.InstalledLanguages**
Application.StartupLanguage

Example

```
Sub Main
  Dim Language as Integer
  Language = Session.Language
  If Language <> smlENGLISH Then
     MsgBox "Switching the current language to English"
     Session.Lanugage = smlENGLISH
  End If
End Sub
```

See Also Application and Session Features on page 11

Session.LoadKeyboardMap

Syntax **Session.KeyboardMap keymapname$**

where **keymapname$** is the name of the keyboard map to load (string).

Description Loads a keyboard map and returns the operation's completion status (boolean). To load the default keyboard map, specify the string **""**.

Example

```
Sub Main
  If Session.LoadKeyboardMap("Keymap1") = FALSE Then
     Session.Echo "Error loading Keymap1, restoring default."
     Session.LoadKeyboardMap "<DEFAULT>"
  End If
End Sub
```

See Also Application and Session Features on page 11

Session.LoadSmarTermButtons

Syntax **Session.LoadSmarTermButtons palettename**

where **palettename** is the name of the SmarTerm Buttons palette to load (string).

Description Loads and displays a SmarTerm Buttons palette and returns the operation’s completion status (boolean). This palette name is optional. If you omit it, the palette associated with the session is loaded.

Example

```
Sub Main
  If Session.LoadSmarTermButtons("c:\SmarTerm\Buttons\toolbar.bpx") = FALSE Then
     MsgBox "Error loading SmarTerm Buttons"
  End If
End Sub
```

See Also Application and Session Features on page 11; User Interaction on page 16

Session.LockStep (object)

Syntax **`Session.LockStep`**

Description Activates the **`LockStep`** state to regulate emulator data flow for the **`Collect, EventWait,`** and **`StringWait`** features (object). The **`Session.Collect, Session.EventWait,`** and **`Session.StringWait`** features are useful when you need to synchronize macro operations with host operations. For example, the macro below uses **`StringWait`** to automate the process of connecting to a host:

```
' A login macro, without LockStep
 Sub Main
   Session.StringWait.MatchString "Username: "
   Session.StringWait.Start
   Session.Send "MyName" + Chr$(13)
   Session.StringWait.MatchString "Password: "
   Session.StringWait.Start
   Session.Send "MyPassword" + Chr$(13)
 End Sub
```

Certain timing problems can, however, prevent a macro such as this from operating reliably. If the host's responsiveness is significantly better than that of your local machine, it would be possible for the **`Session.Send "MyName" + Chr$(13)`** statement to elicit the **`"Password: "`** prompt from the host before the subsequent macro statement, the **`StringWait`**, has been executed. Some, or all, of the **`"Password:"`** string's characters could be processed through the emulator before the **`StringWait`** feature has a chance to begin watching for this string.

The **`LockStep`** feature addresses this timing problem. Here is the login macro again, with **`LockStep`** included:

```
' A login macro, with LockStep
 Sub Main
   Dim LockSession as Object
   Set LockSession = Session.LockStep
   LockSession.Start
   Session.StringWait.MatchString "Username: "
   Session.StringWait.Start
   Session.Send "MyName" + Chr$(13)
   Session.StringWait.MatchString "Password: "
   Session.StringWait.Start
   Session.Send "MyPassword" + Chr$(13)
   Set LockSession = Nothing
 End Sub
```

When the **LockStep** state is active, data arriving from the host is not processed by the emulator until any **EventWait**, **StringWait** or **Collect** macro statements have had a chance to parse that data for match strings. **EventWait**, **StringWait** and **Collect** are 'privileged' against the **LockStep** state to support synchronized data collection.

To instigate the **LockStep** state, it is necessary to assign the return value from **Session.LockStep** to an object pointer and to then use this object point to call the **LockStep** object's **Start** method. Calling the **Start** method without its optional parameter starts a **LockStep** state that persists until it is explicitly deactivated. It is also possible to supply a parameter to this method that specifies the number of seconds that the **LockStep** state should remain in effect. For example, the statements below will activate a **LockStep** state for 12 seconds:

```
Dim L as Object
Set L = Session.LockStep
L.Start 12
```

This state remains in effect until either the **Reset** method is called, the object pointer is assigned the special value of **Nothing**, the object variable goes out of scope, or the macro is halted (e.g. by terminating a debugging session). Note that it will not work to access the **Start** method directly, you must assign the return value of **Session.LockStep** to an object variable and then access the **Start** method through that object variable.

As an example of how **LockStep** is important for use with **Session.Collect**, consider the case where it is necessary for your macro to watch for a **"StartOfMessage"** tag from the host, and then collect all subsequent data until an "**EndOfMessage"** tag is detected. Without **LockStep**, this would look like:

```
'! Collect after StringWait, no LockStep
 Sub Main
   Session.StringWait.MatchString "StartOfMessage"
   Session.StringWait.Start
   Session.Collect.TermString = "EndOfMessage"
   Session.Collect.Start
 End Sub
```

Without the **LockStep** feature, the emulator may process the first portion of the message data before the **Collect** statement is executed. To prevent data loss, **LockStep** can be applied as follows:

```
'! Collect after StringWait, with LockStep
 Sub Main
   Dim L as Object
   Set L = Session.LockStep
   L.Start
   Session.StringWait.MatchString "StartOfMessage"
   Session.StringWait.Start
   Session.Collect.TermString = "EndOfMessage"
   Session.Collect.Start
   L.Reset
 End Sub
```

Example See the examples in the Comments section above.

See Also Host Connections on page 7; Application and Session Features on page 11; Objects on page 18

Session.LockStep.Reset

Syntax **`Session.LockStep.Reset`**

Description Deactivates a **`LockStep`** state.

Example See the examples shown for Session.LockStep (object).

See Also Application and Session Features on page 11

Session.LockStep.Start

Syntax **`Session.LockStep.Start [seconds]`**

where **`seconds`** is the number of seconds that the **`LockStep`** state should last (optional) (integer).

Description Activates a **`LockStep`** state. To instigate a **`LockStep`** state, it is necessary to assign the return value from **`Session.LockStep`** to an object pointer and to then use this object point to call the **`LockStep`** object's Start method. Calling the Start method without its optional parameter starts a **`LockStep`** state that persists until it is explicitly deactivated. It is also possible to supply a parameter to this method that specifies the number of seconds that the **`LockStep`** state should remain in effect.

Note It will not work to access the Start method directly. You must assign the return value of **`Session.LockStep`** to an object variable and then access the Start method through that object variable.

Example See the examples shown for Session.LockStep (object).

See Also Application and Session Features on page 11

Session.MouseCol

Not available for Wyse sessions

Syntax **`Session.MouseCol`**

Description Returns the column of the current mouse position in SmarTerm's session window (integer).

Example

```
Sub Main
  Dim mr as Integer
  Dim mc as Integer

  mr = Session.MouseRow
  mc = Session.MouseCol
  MsgBox "Mouse cursor is on Row: " & Str(mr) & " Column: " & Str(mc)
End Sub
```

See Also Application and Session Features on page 11

Session.MouseRow

Not available for Wyse sessions

Syntax `Session.MouseRow`

Description Returns the row of the current mouse position (integer).

Example

```
Sub Main
  Dim mr as Integer
  Dim mc as Integer

  mr = Session.MouseRow
  mc = Session.MouseCol
  MsgBox "Mouse cursor is on Row: " & Str(mr) & " Column: " & Str(mc)
End Sub
```

See Also Application and Session Features on page 11

Session.NativeScreenText

3270 and 5250 sessions only

Syntax `Session.NativeScreenText(startrow, startcol, endrow, endcol)`

Description Returns the specified screen text from SmarTerm's terminal window, in EBCDIC (string). Parameters are:

Parameter	Description
startrow	The starting row of the text to retrieve.
startcol	The starting column of the text to retrieve.
Endrow	The ending row of the text to retrieve.
Endcol	The ending column of the text to retrieve.

If any parameter has a value of 0, the row or column used is either the first or last (start and end respectively). Field marks are replaced by null characters. Any values out of bounds are truncated to the end of the display buffer.

Example

```
Sub Main
  Dim strText as String
' Read screen from row 4, column 11 through row 5, column 20
  strText = Session.NativeScreenText(4, 11, 5, 20)
End Sub
```

See Also Application and Session Features on page 11

Session.Normal

Syntax `Session.Normal`

VT, SCO, ANSI, and DG sessions only

Description Returns or sets the normal attribute of SmarTerm's display presentation (boolean)

Example

```
Sub Main
  Dim NormState as Boolean
  NormState = Session.Normal
  Session.Normal = True
End Sub
```

See Also Application and Session Features on page 11

Session.Online

Syntax `Session.Online`

Description Returns or sets the status of the session's online state (boolean).

Example

```
Sub Main
  Dim OnLineState as Boolean
  OnLineState = Session.OnLine
  If OnLineState = FALSE Then
     Session.Echo "Cannot continue because you are offline"
     Session.Online = TRUE
  End If
End Sub
```

Session.Page

VT and SCO sessions only

Syntax `Session.Page`

Description Returns or sets the current page in SmarTerm's active session type (integer).

Example

```
Sub Main
  Dim PageNumber as Integer
  PageNumber = Session.Page
   Session.Page = PageNumber + 1
End Sub
```

See Also Application and Session Features on page 11

Session.ReplayCaptureFile

Syntax `Session.ReplayCaptureFile "<captured filename and path>"`

Description Replays the specified SmarTerm capture file. The filename parameter must have quotes around it. If no file name is specified, the Replay captured file dialog is opened. The filename parameter may also contain the path to the file. If no path is specified, SmarTerm looks in the SmarTerm transfer folder. If the path/filename does not exist, the Session.ReplayCaptureFile command is ignored.

Examples Brings up the Replay captured file dialog:

```
Session.ReplayCaptureFile ""
```

Replays the file capture called file.cap. It assumes the file is in the SmarTerm transfer folder:

```
Session.ReplayCaptureFile "file.cap"
```

Replays the file file.cap located in c:\temp:

```
Session.ReplayCaptureFile "c:\temp\file.cap"
```

See Also Application and Session Features on page 11

Session.Row

Syntax `Session.Row`

Description Returns or sets where the cursor is placed in the active SmarTerm session window (integer).

Example

```
Sub Main
  Dim CurrentRow as Integer
  CurrentRow = Session.Row
  Session.Row = CurrentRow + 1
End Sub
```

See Also Application and Session Features on page 11

Session.ScreenText

Syntax `Session.ScreenText(row, column, page, chars)`

Description Returns the specified screen text from SmarTerm's terminal window (string). Parameters are:

Parameter	Description
`row`	The row of the text to retrieve.
`column`	The column of the text to retrieve.
`page`	The page of the text to retrieve.
`chars`	The number of characters to retrieve.

Example

```
Sub Main
  Dim ScnText as String

  ScnText = Session.ScreenText(4, 11, 1, 12)
  Session.Echo ScnText
End Sub
```

See Also Application and Session Features on page 11

Session.ScreenToFile

Syntax `Session.ScreenToFile(filename$)`

where **`filename$`** is the name of the file in which to write the screen data (string).

Description Returns the completion status of the screen capture (boolean). This method captures all text pages and places them in the ASCII text file named with **`filename$`**. Each time this method is called with the same filename, the previous file is overwritten.

Example

```
Sub Main
  Dim RetVal as Boolean
  RetVal = Session.ScreenToFile("scntext.txt")
  If RetVal = False Then
     Session.Echo "An Error Occurred"
  End If
End Sub
```

See Also Drive, Folder, and File Access on page 4; Application and Session Features on page 11

Session.SelectScreenAtCoords

Syntax **`Session.SelectScreenAtCoords(top%, left%, bottom%, right%)`**

Description Selects the text within the boundaries set by **`top%`**, **`left%`**, **`bottom%`**, and **`right%`**. If the selection is successful this method returns **`True`**. Otherwise, it returns **`False`**.

Note This method is not supported in graphics mode emulation.

Parameter	Description
`top%`	The top row of the text to select.
`left%`	The left column of the text to select.
`bottom%`	The bottom row of the text to select.
`right%`	The right column of the text to select.

Example

```
'This example sets the selection and reports its success
Sub Main
   SelectedText = Session.SelectScreenAtCoords(0, 0, 10, 10)
  If SelectedText Then
    ScnText$ = Session.ScreenText(0,0,1,10)
    MsgBox("Selected text: " & ScnText$)
  Else
    MsgBox("Nothing to select.")
  End If
End Sub
```

See Also Application and Session Features on page 11

Session.SelectionEndColumn

Syntax **`Session.SelectionEndRow`**

Description Returns or sets the ending column of the selection (integer). This property is an element of the quartet that also includes **`Session.SelectionStartRow`**, **`Session.SelectionStartColumn`**, and

Session.SelectionEndRow. The text selection is not marked until all four elements have been set so as to define a valid selection. If there is no selection, or if the four elements define an invalid selection box, this property returns **-1**.

Note This method is not supported in graphics mode emulation.

Example

```
'This example selects the entire screen, using the Session
' object to determine the size of the screen.
Sub Main
  MsgBox("Selecting entire screen.")
  Session.SelectionStartRow = 0
  Session.SelectionStartColumn = 0
  Session.SelectionEndRow = Session.TotalRows
  Session.SelectionEndColumn = Session.TotalColumns
End Sub
```

See Also Application and Session Features on page 11

Session.SelectionEndRow

Syntax **Session.SelectionEndRow**

Description Returns or sets the ending row of the selection (integer). This property is an element of the quartet that also includes **Session.SelectionStartRow**, **Session.SelectionStartColumn**, and **Session.SelectionEndColumn**. The text selection is not marked until all four elements have been set so as to define a valid selection. If there is no selection, or if the four elements define an invalid selection box, this property returns **-1**.

Note This method is not supported in graphics mode emulation.

Example

```
'This example selects the entire screen, using the Session
' object to determine the size of the screen.
Sub Main
  MsgBox("Selecting entire screen.")
  Session.SelectionStartRow = 0
  Session.SelectionStartColumn = 0
  Session.SelectionEndRow = Session.TotalRows
  Session.SelectionEndColumn = Session.TotalColumns
End Sub
```

See Also Application and Session Features on page 11

Session.SelectionStartColumn

Syntax **Session.SelectionStartColumn**

Description Returns or sets the starting column of the selection (integer). This property is an element of the quartet that also includes **Session.SelectionStartRow**, **Session.SelectionEndRow**, and **Session.SelectionEndColumn**. The text selection is not marked until all four elements have been set so as to define a valid selection. If there is no selection, or if the four elements define an invalid selection box, this property returns **-1**.

Note This method is not supported in graphics mode emulation.

Example

```
'This example selects the entire screen, using the Session
' object to determine the size of the screen.
Sub Main
  MsgBox("Selecting entire screen.")
  Session.SelectionStartRow = 0
  Session.SelectionStartColumn = 0
  Session.SelectionEndRow = Session.TotalRows
  Session.SelectionEndColumn = Session.TotalColumns
End Sub
```

See Also Application and Session Features on page 11

Session.SelectionStartRow

Syntax **`Session.SelectionStartRow`**

Description Returns or sets the starting row of the selection (integer). This property is an element of the quartet that also includes **`Session.SelectionStartColumn`**, **`Session.SelectionEndRow`**, and **`Session.SelectionEndColumn`**. The text selection is not marked until all four elements have been set so as to define a valid selection. If there is no selection, or if the four elements define an invalid selection box, this property returns **`-1`**.

Note This method is not supported in graphics mode emulation.

Example

```
'This example selects the entire screen, using the Session
' object to determine the size of the screen.
Sub Main
  MsgBox("Selecting entire screen.")
  Session.SelectionStartRow = 0
  Session.SelectionStartColumn = 0
  Session.SelectionEndRow = Session.TotalRows
  Session.SelectionEndColumn = Session.TotalColumns
End Sub
```

See Also Application and Session Features on page 11

Session.SelectionRectangular

Syntax **`Session.SelectionRectangular`**

Description Returns or sets whether or not the selection is rectangular (Boolean). If this property is **`True`**, the selection is rectangular, selecting a block of text. If the property is **`False`**, the selection is linear, selecting text line by line.

Note This method is not supported in graphics mode emulation.

Example

```
'This example toggles the selection between rectangular and
' linear, regardless of the current setting.
Sub Main
  RectSel = Session.SelectionRectangular
```

```
  If RectSel Then
    MsgBox("Selection is rectangular. Changing to linear.")
  Else
    MsgBox("Selection is linear. Changing to rectangular.")
  End If
  RectSel = Not RectSel
End Sub
```

See Also Application and Session Features on page 11

Session.SelectionType

Syntax `Session.SelectionType`

Description Returns the status of the selection (integer). If `Session.SelectionType` is `0` (zero), then there is no selection. If it is `1`, then the selection is text.

Note This method is not supported in graphics mode emulation.

Example

```
'This displays the setting of the selection type.
Sub Main
  fSel= Session.SelectScreenAtCoords(0,0,10,10)
  If Session.SelectionType = 0 Then
    MsgBox("Nothing selected.")
  Else
    MsgBox("Something selected.")
  End If
End Sub
```

See Also Application and Session Features on page 11

Session.Send

Syntax `Session.Send text$`

where `text$` is the text to send (string).

Description Sends text to the host. 8-bit to 7-bit control mapping is performed before the string is sent when operating in a 7-bit controls environment.

Note IBM 3270 and 5250 session do not support the use of key mnemonics (such as `<F1>`) with this command. To send keystrokes to an IBM 3270 or 5250 host, use Session.SendKey.

Example

```
Sub Main
  Session.Send "Mail" & Chr$(13)
  Session.Send "Read NewMail<CR><LF>"
End Sub
```

See Also Character and String Manipulation on page 3; Application and Session Features on page 11; Session.SendKey on page 444

Session.SendKey

3270 and 5250 sessions only

Syntax `Session.SendKey key$`

where **`key$`** is a special SmarTerm function to send (string).

Description Sends a special code to the host. Supported functions are marked with an X in the following table.

Function	3270 Support	5250 Support
ALTCURSOR	X	
ATTN	X	X
BLINKCURSOR	X	
BLUE	X	
BS	X	X
BTAB	X	X
CLEAR	X	X
CLICK	X	
CURSORDOWN	X	X
CURSORLEFT	X	X
CURSORRIGHT	X	X
CURSORUP	X	X
DELETE	X	X
DELETEWORD	X	
DUP	X	X
ENTER	X	X
ERASEEOF	X	
ERASEFIELD	X	
ERASEINPUT	X	X
EXTSEL	X	
FIELDCOLOR	X	
FIELDHILIGHT	X	
FM	X	
FTAB	X	X
GREEN	X	
HOME	X	X
INSERT	X	X
NEWLINE	X	X

Function	3270 Support	5250 Support
PA1	X	
PA2	X	
PA3	X	
PF1 through PF24	X	X
PINK	X	
RED	X	
REVERSE	X	
SELATTR	X	
SYSREQ	X	X
TNRESET	X	X
TREQ	X	X
TURQ	X	
UNDERSCORE	X	
WHITE	X	
YELLOW	X	

Example

```
Sub Main
  Session.SendKey "CURSORDOWN"
End Sub
```

See Also Application and Session Features on page 11

Session.SendLiteral

Syntax **`Session.SendLiteral text$`**

where **`text$`** is the text to send (string).

Description Sends text to the host without character translation. The string expression is sent to the host untranslated. 8-bit to 7-bit control mapping is performed before the string is sent when operating in a 7-bit controls environment.

Example

```
Sub Main
  Session.SendLiteral "Read Newmail"
End Sub
```

See Also Application and Session Features on page 11

Session.SetFontSize

Syntax **`Session.SetFontSize width% height%`**

Description Sets the font size of the characters appearing in the SmarTerm session window. Parameters are:

Parameter	Definition
width%	The font width (integer).
height%	The font height (integer)

If either the width or height parameter is set to 0, the auto-fontsize state will be established.

Example

```
Sub Main
  Session.SetFontSize 6, 10
End Sub
```

See Also Application and Session Features on page 11

Session.SetHotSpotsFile

Syntax `Session.SetHotSpotsFile(Filename)`

Description Loads the HotSpot file specified with Filename (string), returning **TRUE** if successful, **FALSE** if the specified file could not be found or if it contains an error. If you specify an empty string, this method unloads the current HotSpot file.

Filename can specify the complete path to the desired HotSpots file. If no path is specified, the program looks in the User HotSpot folder.

If **Session.SetHotSpotsFile** returns **FALSE**, the original HotSpots file should remain loaded. However, your code should always check, as shown in the example below.

Example

```
'This example loads the HotSpot file 3270_A.HOT.
Sub Main
  FileToLoad$= "3270_A.HOT"

' Check to see if we need to load the file.
  If Session.HotSpotsFileName <> FileToLoad$ Then

' Now load the file, checking for success
    If Session.SetHotSpotsFile(FileToLoad$)= TRUE Then

' Success!
      MsgBox FileToLoad$ & " now loaded."

' Uh-oh, didn't work. Determine whether anything is loaded
' and tell user.
    Else
      MsgBox "Unable to load " & FileToLoad$
      CurrentFile$= Session.HotSpotsFileName
      If CurrentFile$ <> "" Then
        MsgBox CurrentFile$ & " still loaded."
      Else
        MsgBox "No HotSpots loaded."
      End If
    End If
  End If
End Sub
```

See Also Application and Session Features on page 11; User Interaction on page 16

Session.StringWait (object)

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.StringWait`**

Description Returns an object supporting access to SmarTerm's **`StringWait`** feature. The **`Session.StringWait`** object is used to wait for specific data to arrive from the host. There is one **`StringWait`** object per-session. Its methods and properties can be divided into three categories: those used to initialize the wait object, those used to activate a wait, and those used to check the results of the wait. These categories are as follows:

Initialization
Session.StringWait.Reset
Session.StringWait.MatchString
Session.StringWait.MatchStringExact
Session.StringWait.MatchStringEx
Session.StringWait.Timeout
Session.StringWait.TimeoutMS
Session.StringWait.MaxCharacterCount

Activation Session.StringWait.Start

Results Session.StringWait.Status

The **`StringWait`** object automatically resets to its default (empty) state the first time any of its properties is set or any of its methods called after a previous **`StringWait`** operation has completed.

In certain cases, it may be necessary to use the **`Lockstep`** feature to insure that the **`StringWait`** object is presented with all data from the host that is significant. See the discussion of **`Session.Lockstep`** for further details.

Example

```
Sub Main
  ' Simple StringWait -- a single match string
  Session.StringWait.MatchString "Login: "
  Session.StringWait.Start
  if Session.StringWait.Status = 1 Then
     Session.Echo "Match string detected"
  End If
  ' Multiple match strings -- where the order of the
  ' MatchString calls define the ordinals.
  Dim MatchOrdinal as integer
  Session.StringWait.MatchString "One"
  Session.StringWait.MatchString "Two"
  Session.StringWait.MatchString "Three"
  MatchOrdinal = Session.StringWait.Start
  Select Case MatchOrdinal
    Case 1
        Session.Echo "Detected a One"
    Case 2
```

```
        Session.Echo "Detected a Two"
    Case 3
        Session.Echo "Detected a Three"
  End Select
  ' Using MatchStringEx, a timeout, and a max character count
  Session.StringWait.MatchStringEx "One", TRUE, 3
  Session.StringWait.MatchStringEx "Two", FALSE, 5
  Session.StringWait.Timeout = 25
  Session.StringWait.MaxCharacterCount = 10
  MatchOrdinal = Session.StringWait.Start
  Select Case MatchOrdinal
    Case 3
        Session.Echo "Detected a One"
    Case 5
        Session.Echo "Detected a Two"
    Case smlWAITTIMEOUT
        Session.Echo "Timeout expired"
    Case smlWAITMAXCHARS
        Session.Echo "Max characters exceeded"
  End Select
End Sub
```

See Also Character and String Manipulation on page 3; Application and Session Features on page 11; Objects on page 18

Session.StringWait.MatchString

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.StringWait.MatchString(pattern_string)`**

where **`pattern_string`** is the string to register for match detection.

Description Registers a match pattern with the **`StringWait`** object. When the **`StringWait`** operation is started, using its **`Start`** method, it will be terminated when a match is detected with a registered string in the host-to-terminal data stream. Returns an integer that indicates the ordinal value associated with the registered string.

The comparison is case-insensitive. If case sensitivity is desired, use the **`MatchStringExact`** method instead. The value returned by the method is the ordinal number that will be returned by the **`Start`** method (and subsequently, the **`Status`** property) if this is the pattern which terminates the **`StringWait`** operation. Note that it is not necessary to record this ordinal if you take advantage of the fact that the first pattern string registered will be ordinal **`1`**, the second will be ordinal **`2`**, etc.

Example See the examples under Session.StringWait (object).

See Also Application and Session Features on page 11

Session.StringWait.MatchStringEx

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.StringWait.MatchStringEx(pattern_string, case_sense, ordinal)`**

Description Registers a match pattern with the **`StringWait`** object. When the **`StringWait`** operation is started, using its **`Start`** method, it will be terminated when a match is detected with a registered string in the host-to-terminal data stream. Returns an integer that indicates the ordinal value associated with the registered string. Parameters are:

Parameter	Description
`pattern_string`	The string to register for match detection (string).
`case_sense`	The comparison is case-sensitive if the second parameter is **`True`** (boolean).
`Ordinal`	The ordinal value of the match pattern is specified by the third parameter. If this is <= 0, the ordinal value of the string is set to one greater than the largest ordinal value assigned so far (integer).

Multiple match patterns can share a single ordinal value. The value returned by the method is the ordinal number that will be returned by the Start method (and subsequently, the **`Status`** property) if this is the pattern which terminates the **`StringWait`** operation. Note that it is not necessary to record this ordinal since the value returned will be that specified as the "ordinal" entry parameter.

Example See the examples under Session.StringWait (object).

See Also Application and Session Features on page 11

Session.StringWait.MatchStringExact

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.StringWait.MatchStringExact(pattern_string)`**

where **`pattern_string`** is the string to register for match detection.

Description Registers a match pattern with the **`StringWait`** object. When the **`StringWait`** operation is started, using its **`Start`** method, it will be terminated when a match is detected with a registered string in the host-to-terminal data stream. Returns an integer that indicates the ordinal value associated with the registered string.

The comparison is case-sensitive. If case insensitivity is desired, use the **`MatchString`** method instead. The value returned by the method is the ordinal number that will be returned by the **`Start`** method (and subsequently, the **`Status`** property) if this is the pattern which terminates the **`StringWait`** operation. Note that it is not necessary to record this ordinal if you take advantage of the fact that the first pattern string registered will be ordinal **`1`**, the second will be ordinal **`2`**, etc.

Example See the examples under Session.StringWait (object).

See Also Application and Session Features on page 11

Session.StringWait.MaxCharacterCount

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.StringWait.MaxCharacterCount`**

Description Sets the maximum number of characters to **`StringWait`** before the **`StringWait`** operation terminates.

Example See the examples under Session.StringWait (object).

See Also Application and Session Features on page 11

Session.StringWait.Reset

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.StringWait.Reset`**

Description Resets the wait object's properties to their default values. The **`StringWait`** object automatically resets to its default (empty) state when any of its properties is set or any of its methods called after a previous **`StringWait`** operation has completed.

Example See the examples under Session.StringWait (object).

See Also Application and Session Features on page 11

Session.StringWait.Start

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.StringWait.Start`**

Description Returns a status value that indicates the reason that the wait ended (integer). This method activates the wait object, returning only when the specified conditions have been met. The status of the **`StringWait`** operation is returned by the object's **`Start`** method and is also available through its **`Status`** property. The possible values are shown in the table below.

Value	Constant	Meaning
>=1	N/A	Ordinal indicating successful match (see below)
-1	**`smlWAITTIMEOUT`**	Timeout
-2	**`smlWAITMAXCHARS`**	Maximum characters
-15	**`smlWAITERROR`**	Miscellaneous error

The value returned in the case of a match is the ordinal corresponding to the string which was matched. This ordinal is determined by the method chosen to register the match strings. When either the **`MatchString`** or **`MatchStringExact`** methods are used, the ordinal is determined by the sequence of the calls made to these methods. When the **`MatchStringEx`** method is used, the ordinal is determined by the caller, as an entry parameter to the method call. See the Comments for these methods for further details.

Example See the examples under Session.StringWait (object).

See Also Application and Session Features on page 11

Session.StringWait.Status

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.StringWait.Status`**

Description Returns the most recent value returned by the **`Start`** method, or 0 if the wait object has been reset (integer). The status of the **`StringWait`** operation is returned by the object's **`Start`** method and is also available through its **`Status`** property. The possible values are shown in the table below.

Value	Constant	Meaning
>=1	N/A	Ordinal indicating successful match (see below)
-1	**`smlWAITTIMEOUT`**	Timeout
-2	**`smlWAITMAXCHARS`**	Maximum characters
-15	**`smlWAITERROR`**	Miscellaneous error

The value returned in the case of a match is the ordinal corresponding to the string which was matched. This ordinal is determined by the method chosen to register the match strings. When either the **`MatchString`** or **`MatchStringExact`** methods are used, the ordinal is determined by the sequence of the calls made to these methods. When the **`MatchStringEx`** method is used, the ordinal is determined by the caller, as an entry parameter to the method call. See the Comments for these methods for further details.

Example See the examples under Session.StringWait (object).

See Also Application and Session Features on page 11

Session.StringWait.Timeout

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.StringWait.Timeout`**

Description Sets the maximum number of seconds to allow for the **`StringWait`** operation. This property is read-write.

Example See the examples under Session.StringWait (object).

See Also Application and Session Features on page 11

Session.StringWait.TimeoutMS

VT, SCO, ANSI, and DG sessions only

Syntax `Session.StringWait.TimeoutMS`

Description Sets the maximum number of milliseconds to allow for the **`StringWait`** operation. This property is read-write.

Example See the examples under Session.StringWait (object).

See Also Application and Session Features on page 11

Session.TotalColumns

Syntax `Session.TotalColumns`

Description Returns the total number of columns available in the active SmarTerm session (integer).

Example

```
Sub Main
  Dim Cols as Integer
  Cols = Session.TotalColumns
  If Cols <> 132 Then
     Session.Echo "This application will not run correctly unless " & _
            "you are in 132 column mode"
  End If
End Sub
```

See Also Application and Session Features on page 11

Session.TotalPages

Syntax `Session.TotalPages`

Description Returns the total number of pages available in the active session (integer).

Example

```
Sub Main
  Dim Pages as Integer
  Pages = Session.TotalPages
  Session.Echo "This emulation type supports " & Pages & " pages."
End Sub
```

See Also Application and Session Features on page 11

Session.TotalRows

Syntax `Session.TotalRows`

Description Returns the total number of rows available in the active session (integer).

Example

```
Sub Main
  Dim Rows as Integer
  Rows = Session.TotalRows
```

```
  If Rows <> 24 Then
     Session.Echo "Please set number of rows to 24"
  End If
End Sub
```

See Also Application and Session Features on page 11

Session.Transfer

Syntax **`Session.Transfer`**

Description Returns the Transfer object for the session. The **`Session.Transfer`** property is intended for use by external VBA controllers. The predefined Transfer object exists for use by internal macros.

Example

```
Dim MyTransfer as Object
MyTransfer = Session.Transfer
```

Session.TransferProtocol

Syntax **`Session.TransferProtocol(protocolname)`**

Description Sets the file transfer protocol in the active SmarTerm session, returning the operation's completion status (boolean). **`protocolname`** is the name of the new file transfer protocol to establish (string). Possible values are:

```
FTP
KERMIT
XMODEM
YMODEM
ZMODEM
IND$FILE
```

Example

```
Sub Main
  Dim RetVal as Boolean
  RetVal = Session.TransferProtocol("XMODEM")
  If RetVal Then
     Session.Echo "Protocol set to XMODEM"
  Else
     Session.Echo "Unable to set protocol to XMODEM"
  End If
End Sub
```

See Also File Transfer on page 2; Application and Session Features on page 11; Objects on page 18

Session.TranslateBinary

Syntax **`Session.TranslateBinary`**

Description Returns or sets whether character translation is applied by file transfers of binary files (boolean).

Note This property does not apply to IND$FILE transfers, or to text file transfers such as those with the **`Session.Capture`**, **`Session.TransmitFile`**, or **`Session.TransmitFileUntranslated`** methods.

Example

```
Sub Main
  Session.TranslateBinary = True
  Transfer.SendFile "ToHost.txt"
End Sub
```

See Also File Transfer on page 2; Application and Session Features on page 11

Session.TranslateText

Syntax `Session.TranslateText`

Description Returns or sets whether character translation from the host format to the PC format is applied by **`Session.Capture`** and **`Session.TransmitFile`** (boolean).

Note This property does not apply to IND$FILE, where all translation is done in ANSI or ASCII. Neither does it affect the translation of character mnemonics to actual characters (such as "<CR>" to a carriage return), which is handled by the choice of the **`Session.Transmit`** method (translated) or the **`Session.TransmitFileUntranslated`** method (not translated).

Example

```
Sub Main
  Session.TranslateText = True
  Session.TransmitFile "ToHost.txt"
End Sub
```

See Also File Transfer on page 2; Application and Session Features on page 11

Session.TransmitFile

Syntax `Session.TransmitFile(filename$)`

where **`filename$`** is the name of the file to send to the host (string).

Description Returns the operation's completion status (boolean). Sends the specified ASCII file to the host, translating character mnemonics into the actual characters (such as "<CR>" to a carriage return). If you do not want this character translation to occur, use the **`Session.TransmitFileUntranslated`** method.

Note The translation of characters from PC format to host format is controlled by the setting of the **`Session.TranslateText`** property.

Example

```
Sub Main
  Dim RetVal as Boolean
 'Create the file on a VAX host.
  Session.Send "create DataFile.Txt<CR>"
  Sleep 2000
 'Start sending the file.
  RetVal = Session.TransmitFile("<path to valid text file>")
  If RetVal = True Then
     Session.Send "^Z"
  Else
     Session.Send "^Y"
```

```
    Session.Echo "An error occurred transmitting the file."
  End If
End Sub
```

See Also File Transfer on page 2; Application and Session Features on page 11

Session.TransmitFileUntranslated

Syntax **`Session.TransmitFileUntranslated(filename$)`**

where **`filename$`** is the name of the file to send to the host (string).

Description Returns the operation's completion status (boolean). Sends the specified ASCII file to the host without translating character mnemonics into the actual characters (such as "<CR>" to a carriage return). If you do want this character translation to occur, use the **`Session.TransmitFile`** method.

Note The translation of characters from PC format to host format is controlled by the setting of the **`Session.TranslateText`** property.

Example

```
Sub Main
  Dim RetVal as Boolean
 'Create the file on a VAX host.
  Session.Send "create DataFile.Txt<CR>"
  Sleep 2000
 'Start sending the file.
  RetVal = Session.TransmitUntranslated("c:\DataFile.Txt")
  If RetVal = True Then
    Session.Send "^Z"
  Else
    Session.Send "^Y"
    Session.Echo "An error occurred transmitting the file."
  End If
End Sub
```

See Also File Transfer on page 2; Application and Session Features on page 11

Session.TriggersActive

Syntax **`Session.TriggersActive`**

Description Sets or returns the state of the Triggers feature (Boolean). If set to TRUE then Triggers are active; if set to FALSE then Triggers are turned off.

Example

```
Sub Main

If Session.TriggersActive = TRUE Then Then
   MsgBox "Triggers now on. Turning Triggers off."
   Session.TriggersActive = FALSE
Else
MsgBox "Triggers now off. Turning Triggers on."
   Session.TriggersActive = TRUE
End If

End Sub
```

See Also Application and Session Features on page 11

Session.TypeFile

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.TypeFile(filename$)`**

`where filename$` is the name of the file to send to the display (string).

Description Returns the operation's completion status (boolean). Displays file's contents on the screen as if it had been sent by the host.

Example

```
Sub Main
  Dim RetVal as Boolean
  RetVal = Session.TypeFile("c:\DataFile.Txt")
  If RetVal = False Then
     Session.Echo "An error occurred"
  End If
End Sub
```

Session.Underline

VT, SCO, ANSI, and DG sessions only

Syntax **`Session.Underline`**

Description Returns or sets the underline attribute of the display presentation (boolean)

Example

```
Sub Main
  Dim Underline State as Boolean
  Underline State = Session.Underline
  Session.Underline = True
End Sub
```

See Also Application and Session Features on page 11

Session.UnloadSmarTermButtons

Syntax **`Session.UnloadSmarTermButtons`**

Description Unloads and hides a palette associated with the session and returns the operation's completion status (boolean).

Example

```
Sub Main
  If Session.UnloadSmarTermButtons = FALSE Then
     MsgBox "Error unloading SmarTerm Buttons"
  End If
End Sub
```

See Also Application and Session Features on page 11; User Interaction on page 16

Session.Visible

Syntax **`Session.Visible`**

Description Returns or sets the visible state of the SmarTerm session (boolean). This property can be used to make a SmarTerm session invisible.

Example

```
Sub Main
  Dim Visible as Boolean
  Visible = Session.Visible
  Session.Visible = False
End Sub
```

See Also Application and Session Features on page 11

Session.WindowState

Syntax **`Session.WindowState`**

Description Returns or sets a SmarTerm session's window state (integer). Possible values are:

Value	Constant	Meaning
0	**smlMINIMIZE**	The window is minimized.
1	**smlRESTORE**	The window is restored.
2	**smlMAXIMIZE**	The window is maximized.

Example

```
Sub Main
  Dim WinState as Integer
  WinState = Session.WindowState
  If WinState = smlMINIMIZE Then
     Session.WindowState = smlMAXIMIZE
  End If
End Sub
```

See Also Application and Session Features on page 11

Set

Syntax 1 **Set *object_var* = *object_expression***

Syntax 2 **Set *object_var* = New *object_type***

Syntax 3 **Set *object_var* = Nothing**

Description Assigns a value to an object variable.

Syntax 1

The first syntax assigns the result of an expression to an object variable. This statement does not duplicate the object being assigned but rather copies a reference of an existing object to an object variable.

The **`object_expression`** is any expression that evaluates to an object of the same type as the **`object_var`**.

With data objects, **`Set`** performs additional processing. When the **`Set`** is performed, the object is notified that a reference to it is being made and destroyed. For example, the following statement deletes a reference to object A, then adds a new reference to B.

```
Set a = b
```

In this way, an object that is no longer being referenced can be destroyed.

Syntax 2

In the second syntax, the object variable is being assigned to a new instance of an existing object type. This syntax is valid only for data objects.

When an object created using the **`New`** keyword goes out of scope (i.e., the **`Sub`** or **`Function`** in which the variable is declared ends), the object is destroyed.

Syntax 3

The reserved keyword **`Nothing`** is used to make an object variable reference no object. At a later time, the object variable can be compared to **`Nothing`** to test whether the object variable has been instantiated:

```
Set a = Nothing
  :
If a Is Nothing Then Beep
```

Example

```
Sub Main
  Dim document As Object
  Dim page As Object
  Set document = GetObject("c:\resume.doc")
  Set page = Document.ActivePage
  Session.Echo page.name
End Sub
```

See Also Objects on page 18

SetAttr

Syntax **`SetAttr`** ***`pathname, attributes`***

Description Changes the attribute **`pathname`** to the given attribute. A runtime error results if the file cannot be found. The **`SetAttr`** statement accepts the following named parameters:

Parameter	Description
`pathname`	String containing the name of the file.
`Attributes`	Integer specifying the new attribute of the file.

The **attributes** parameter can contain any combination of the following values:

Constant	Value	Includes
ebNormal	0	Turns off all attributes
ebReadOnly	1	Read-only files
ebHidden	2	Hidden files
ebSystem	4	System files
ebVolume	8	Volume label
ebArchive	32	Files that have changed since the last backup
ebNone	64	Files with no attributes

The attributes can be combined using the **+** operator or the binary **Or** operator.

Example

```
Sub Main
  Open "test.dat" For Output Access Write As #1
  Close
  Session.Echo "The current file attribute is: " & GetAttr("test.dat")
  SetAttr "test.dat",ebReadOnly Or ebSystem
  Session.Echo "The file attribute was set to: " & GetAttr("test.dat")
End Sub
```

See Also Drive, Folder, and File Access on page 4

Sgn

Syntax **Sgn(*number*)**

Description Returns an **Integer** indicating whether a number is less than, greater than, or equal to 0. Returns 1 if **number** is greater than 0. Returns 0 if **number** is equal to 0. Returns –1 if **number** is less than 0.

The **number** parameter is a numeric expression of any type. If number is **Null**, then a runtime error is generated. **Empty** is treated as 0.

Example

```
Sub Main
  a% = -100
  b% = 100
  c% = a% * b%
  Select Case Sgn(c%)
    Case -1
      Session.Echo "The product is negative " & Sgn(c%)
    Case 0
      Session.Echo "The product is 0 " & Sgn(c%)
    Case 1
      Session.Echo "The product is positive " & Sgn(c%)
  End Select
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

Shell

Syntax **Shell(*pathname* [,*windowstyle*])**

Description Executes another application, returning the task ID if successful. The **Shell** statement accepts the following named parameters:

Parameter	Description	
pathname	String containing the name of the application and any parameters.	
Windowstyle	Optional integer specifying the state of the application window after execution. It can be any of the following values:	
	ebHide	Application is hidden.
	ebNormalFocus	Application is displayed in default position with the focus.
	ebMinimizedFocus	Application is minimized with the focus (this is the default).
	ebMaximizedFocus	Application is maximized with the focus.
	ebNormalNoFocus	Application is displayed in default position without the focus.
	ebMinimizedNoFocus	Application is minimized without the focus

A runtime error is generated if **windowstyle** is not one of the above values.

An error is generated if unsuccessful running **pathname**.

The **Shell** command runs programs asynchronously: the statement following the **Shell** statement will execute before the child application has exited. The next statement may run even before the child application has finished loading.

The **Shell** function returns a value suitable for activating the application using the **AppActivate** statement.

This function returns a global process ID that can be used to identify the new process. The **Shell** function does not support file associations (i.e., setting pathname to **"sample.txt"** will not execution Notepad).

When specifying long filenames as parameters, you may have to enclose the parameters in double quotes. For example, to run WordPad, passing it a file called "Sample Document", you would use the following statement:

```
r = Shell("WordPad ""Sample Document""")
```

Example

```
Sub Main
  id = Shell("clock.exe",1)
  AppActivate "Clock"
  Sleep(2000)
  AppClose "Clock"
End Sub
```

See Also Operating System Control on page 15

Sin

Syntax `Sin(number)`

Description Returns a **`Double`** value specifying the sine of **`number`**. The **`number`** parameter is a **`Double`** specifying an angle in radians.

Example

```
Sub Main
  c# = Sin(Pi / 4)
  Session.Echo "The sine of 45 degrees is: " & c#
End Sub
```

See Also Tan; Cos; Atn.

Single (data type)

Syntax `Single`

Description Used to declare variables capable of holding real numbers with up to seven digits of precision. Single variables are used to hold numbers within the following ranges:

Sign	Range
Negative	-3.402823E38 <= single <= -1.401298E-45
Positive	1.401298E-45 <= single <= 3.402823E38

The type-declaration character for **`Single`** is **`!`**.

Storage

Internally, singles are stored as 4-byte (32-bit) IEEE values. Thus, when appearing within a structure, singles require 4 bytes of storage. When used with binary or random files, 4 bytes of storage is required.

Each single consists of the following:

- A 1-bit sign
- An 8-bit exponent

- A 24-bit mantissa

See Also Numeric, Math, and Accounting Functions on page 9

Sleep

Syntax **`Sleep`** ***`milliseconds`***

Description Causes the macro to pause for a specified number of milliseconds. The **`milliseconds`** parameter is a **`Long`** in the following range:

`0 <=` ***`milliseconds`*** **`<= 2,147,483,647`**

Example

```
Sub Main
  Msg.Open "Waiting 2 seconds",0,False,False
  Sleep(2000)
  Msg.Close
End Sub
```

Under Windows, the accuracy of the system clock is modulo 55 milliseconds. The value of **`milliseconds`** will, in the worst case, be rounded up to the nearest multiple of 55. In other words, if **`milliseconds`** is 1, it will be rounded to 55 in the worst case.

See Also Macro Control and Compilation on page 10

Sln

Syntax **`Sln(`*`cost`*`, `*`salvage`*`, `*`life`*`)`**

Description Returns the straight-line depreciation of an asset assuming constant benefit from the asset. The **`Sln`** of an asset is found by taking an estimate of its useful life in years, assigning values to each year, and adding up all the numbers. The formula used to find the **`Sln`** of an asset is as follows:

`(Cost - Salvage Value) / Useful Life`

The **`Sln`** function requires the following named parameters:

Parameter	Description
`cost`	Double representing the initial cost of the asset.
`Salvage`	Double representing the estimated value of the asset at the end of its useful life.
`Life`	Double representing the length of the asset's useful life.

The unit of time used to express the useful life of the asset is the same as the unit of time used to express the period for which the depreciation is returned.

Example This example calculates the straight-line depreciation of an asset that cost $10,000.00 and has a salvage value of $500.00 as scrap after ten years of service life.

```
Sub Main
  dep# = Sln(10000.00,500.00,10)
  Session.Echo "The annual depreciation is: " & Format(dep#,"Currency")
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

Space, Space$

Syntax **Space[$](*number*)**

Description Returns a string containing the specified number of spaces. **Space$** returns a **String**, whereas **Space** returns a **String** variant. The **number** parameter is an **Integer** between 0 and 32767.

Example

```
Sub Main
  ln$ = Space$(10)
  Session.Echo "Hello" & ln$ & "over there."
End Sub
```

See Also Character and String Manipulation on page 3

Spc

Syntax **Spc(*numspaces*)**

Description Prints out the specified number of spaces. This function can only be used with the **Print** and **Print#** statements. The **numspaces** parameter is an **Integer** specifying the number of spaces to be printed. It can be any value between 0 and 32767. If a line width has been specified (using the **Width** statement), then the number of spaces is adjusted as follows:

```
numspaces = numspaces Mod width
```

If the resultant number of spaces is greater than width – print_position, then the number of spaces is recalculated as follows:

```
numspaces = numspaces – (width – print_position)
```

These calculations have the effect of never allowing the spaces to overflow the line length. Furthermore, with a large value for column and a small line width, the file pointer will never advance more than one line.

Example

```
Sub Main
  Viewport.Open
  Print "I am"; Spc(20); "20 spaces apart!"
  Sleep (10000)        'Wait 10 seconds.
  Viewport.Close
End Sub
```

See Also Character and String Manipulation on page 3; Drive, Folder, and File Access on page 4

SQLBind

Syntax **`SQLBind(`*`connectionnum`*`, `*`array`*` [,`*`column`*`])`**

Description Specifies which fields are returned when results are requested using the **`SQLRetrieve`** or **`SQLRetrieveToFile`** function. The following table describes the named parameters to the **`SQLBind`** function:

Parameter	Description
`connectionnum`	Long parameter specifying a valid connection.
`Array`	Any array of variants. Each call to SQLBind adds a new column number (an integer) in the appropriate slot in the array. Thus, as you bind additional columns, the **`array`** parameter grows, accumulating a sorted list (in ascending order) of bound columns. If **`array`** is fixed, then it must be a one-dimensional variant array with sufficient space to hold all the bound column numbers. A runtime error is generated if **`array`** is too small. If **`array`** is dynamic, then it will be resized to exactly hold all the bound column numbers.
`Column`	Optional long parameter that specifies the column to which to bind data. If this parameter is omitted, all bindings for the connection are dropped.

This function returns the number of bound columns on the connection. If no columns are bound, then 0 is returned. If there are no pending queries, then calling **`SQLBind`** will cause an error (queries are initiated using the **`SQLExecQuery`** function).

If supported by the driver, row numbers can be returned by binding column 0.

There is a trappable runtime error if **`SQLBind`** fails. Additional error information can then be retrieved using the **`SQLError`** function.

Example This example binds columns to data.

```
Sub Main
  Dim columns() As Variant
  id& = SQLOpen("dsn=SAMPLE",,3)
  t& = SQLExecQuery(id&,"Select * From c:\sample.dbf")
  i% = SQLBind(id&,columns,3)
  i% = SQLBind(id&,columns,1)
  i% = SQLBind(id&,columns,2)
  i% = SQLBind(id&,columns,6)
  For x = 0 To (i% - 1)
    Session.Echo columns(x)
  Next x
  id& = SQLClose(id&)
End Sub
```

See Also SQL Access on page 19

SQLClose

Syntax **`SQLClose(`*`connectionnum`*`)`**

Description Closes the connection to the specified data source. The unique connection ID (**`connectionnum`**) is a **`Long`** value representing a valid connection as returned by **`SQLOpen`**. After **`SQLClose`** is called, any subsequent calls made with the **`connectionnum`** will generate runtime errors.

The **`SQLClose`** function returns 0 if successful; otherwise, it returns the passed connection ID and generates a trappable runtime error. Additional error information can then be retrieved using the **`SQLError`** function.

The compiler automatically closes all open SQL connections when either the macro or the application terminates. You should use the **`SQLClose`** function rather than relying on the compiler to automatically close connections in order to ensure that your connections are closed at the proper time.

Example

```
Sub Main
  id& = SQLOpen("dsn=SAMPLE",,3)
  id& = SQLClose(id&)
End Sub
```

See Also SQL Access on page 19

SQLError

Syntax **`SQLError(`*`resultarray`*`, `*`connectionnum`*`)`**

Description Retrieves driver-specific error information for the most recent SQL functions that failed. This function is called after any other SQL function fails. Error information is returned in a two-dimensional array (**`resultarray`**). The following table describes the named parameters to the **`SQLError`** function:

Parameter	Description
`resultarray`	Two-dimensional variant array, which can be dynamic or fixed. If the array is fixed, it must be (**`x`**,3), where **`x`** is the number of errors you want returned. If **`x`** is too small to hold all the errors, then the extra error information is discarded. If **`x`** is greater than the number of errors available, all errors are returned, and the empty array elements are set to empty. If the array is dynamic, it will be resized to hold the exact number of errors.
`Connectionnum`	Optional long parameter specifying a connection ID. If this parameter is omitted, error information is returned for the most recent SQL function call.

Each array entry in the **`resultarray`** parameter describes one error. The three elements in each array entry contain the following information:

Element	Value
(entry,0)	The ODBC error state, indicated by a long containing the error class and subclass.
(entry,1)	The ODBC native error code, indicated by a long.
(entry,2)	The text error message returned by the driver. This field is string type.

For example, to retrieve the ODBC text error message of the first returned error, the array is referenced as:

***resultarray*(0,2)**

The **SQLError** function returns the number of errors found.

There is a runtime error if **SQLError** fails. (You cannot use the **SQLError** function to gather additional error information in this case.)

Example

```
Sub Main
  Dim a() As Variant
  On Error Goto Trap
  id& = SQLOpen("",,4)
  id& = SQLClose(id&)
  Exit Sub
Trap:
  rc% = SQLError(a)
  If (rc%) Then
    For x = 0 To (rc% - 1)
      Session.Echo "The SQLState returned was: " & a(x,0)
      Session.Echo "The native error code returned was: " & a(x,1)
      Session.Echo a(x,2)
    Next x
  End If
End Sub
```

SQLExecQuery

Syntax **SQLExecQuery(*connectionnum*, *querytext*)**

Description Executes an SQL statement query on a data source. This function is called after a connection to a data source is established using the **SQLOpen** function. The **SQLExecQuery** function may be called multiple times with the same connection ID, each time replacing all results. The following table describes the named parameters to the SQLExecQuery function:

Parameter	Description
connectionnum	Long identifying a valid connected data source. This parameter is returned by the SQLOpen function.
Querytext	String specifying an SQL query statement. The SQL syntax of the string must strictly follow that of the driver.

The return value of this function depends on the result returned by the SQL statement:

SQL Statement	Value
SELECT...FROM	The value returned is the number of columns returned by the SQL statement
DELETE,INSERT,UPDATE	The value returned is the number of rows affected by the SQL statement

There is a runtime error if **SQLExecQuery** fails. Additional error information can then be retrieved using the **SQLError** function.

Example

```
Sub Main
  Dim s As String
  Dim qry As Long
  Dim a() As Variant
  On Error Goto Trap
  id& = SQLOpen("dsn=SAMPLE", s$, 3)
  qry = SQLExecQuery(id&,"Select * From c:\sample.dbf")
  Session.Echo "There are " & qry & " columns in the result set."
  id& = SQLClose(id&)
  Exit Sub
Trap:
  rc% = SQLError(a)
  If (rc%) Then
    For x = 0 To (rc% - 1)
      Session.Echo "The SQLState returned was: " & a(x,0)
      Session.Echo "The native error code returned was: " & a(x,1)
      Session.Echo a(x,2)
    Next x
  End If
End Sub
```

See Also SQL Access on page 19

SQLGetSchema

Syntax **SQLGetSchema(*connectionnum*, *typenum*, [, [*resultarray*] [, *qualifiertext*]])**

Description Returns information about the data source associated with the specified connection. The following table describes the named parameters to the **SQLGetSchema** function:

Parameter	Description
connectionnum	Long parameter identifying a valid connected data source. This parameter is returned by the SQLOpen function.

Parameter	Description
`Typenum`	Integer parameter specifying the results to be returned. The following are the values for this parameter: 1: Returns a one-dimensional array of available data sources. The array is returned in the **`resultarray`** parameter.2: Returns a one-dimensional array of databases (either directory names or database names, depending on the driver) associated with the current connection. The array is returned in the **`resultarray`** parameter. 3: Returns a one-dimensional array of owners (user IDs) of the database associated with the current connection. The array is returned in the **`resultarray`** parameter. 4: Returns a one-dimensional array of table names for a specified owner and database associated with the current connection. The array is returned in the **`resultarray`** parameter. 5: Returns a two-dimensional array (**`n`** by 2) containing information about a specified table. The first element contains the column name. The second element contains the data type of the column 6: Returns a string containing the ID of the current user. 7: Returns a string containing the name (either the directory name or the database name, depending on the driver) of the current database. 8: Returns a string containing the name of the data source on the current connection. 9: Returns a string containing the name of the DBMS of the data source on the current connection (e.g., "FoxPro 2.5" or "Excel Files"). 10: Returns a string vontaining the name of the server for the data source. 11: Returns a string containing the owner qualifier used by the data source (e.g., "owner," "Authorization ID," "Schema").

Parameter	Description
`Typenum (cont).`	12: Returns a string containing the table qualifier used by the data source (e.g., "table," "file"). 13: Returns a string containing the database qualifier used by the data source (e.g., "database," "directory"). 14: Returns a string containing the procedure qualifier used by the data source (e.g., "database procedure," "stored procedure," "procedure").
`Resultarray`	Optional variant array parameter. This parameter is only required for action values 1, 2, 3, 4, and 5. The returned information is put into this array. If **`resultarray`** is fixed and it is not the correct size necessary to hold the requested information, then SQLGetSchema will fail. If the array is larger than required, then any additional elements are erased. If **`resultarray`** is dynamic, then it will be redimensioned to hold the exact number of elements requested.
`qualifiertext`	Optional string parameter required for actions 3, 4, or 5. The values are as follows: 3: The **`qualifiertext`** parameter must be the name of the database represented by ID. 4: The **`qualifiertext`** parameter specifies a database name and an owner name. The syntax for this string is: **`DatabaseName.OwnerName`** 5: The **`qualifiertext`** parameter specifies the name of a table on the current connection.

There is a runtime error if **`SQLGetSchema`** fails. Additional error information can then be retrieved using the **`SQLError`** function.

If you want to retrieve the available data sources (where **`typenum = 1`**) before establishing a connection, you can pass **`0`** as the **`connectionnum`** parameter. This is the only action that will execute successfully without a valid connection.

This function calls the ODBC functions **`SQLGetInfo`** and **`SQLTables`** in order to retrieve the requested information. Some database drivers do not support these calls and will therefore cause the **`SQLGetSchema`** function to fail.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  Dim dsn() As Variant
  numdims% = SQLGetSchema(0,1,dsn)
  If (numdims%) Then
    mesg = "Valid data sources are:" & crlf
```

```
    For x = 0 To numdims% - 1
      mesg = mesg & dsn(x) & crlf
    Next x
  Else
    mesg = "There are no available data sources."
  End If
  Session.Echo mesg
End Sub
```

See Also SQL Access on page 19

SQLOpen

Syntax **SQLOpen(*connectionstr* [, [*outputref*] [, *driverprompt*]])**

Description Establishes a connection to the specified data source, returning a **Long** representing the unique connection ID. This function connects to a data source using a login string (**connectionstr**) and optionally sets the completed login string (**outputref**) that was used by the driver. The following table describes the named parameters to the **SQLOpen** function:

Parameter	Description
connectionstr	String expression containing information required by the driver to connect to the requested data source. The syntax must strictly follow the driver's SQL syntax.
Outputref	Optional string variable that will receive a completed connection string returned by the driver. If this parameter is missing, then no connection string will be returned.
Driverprompt	Integer expression specifying any of the following values: The driver's login dialog is always displayed. The driver's dialog is only displayed if the connection string does not contain enough information to make the connection. This is the default behavior. The driver's dialog is only displayed if the connection string does not contain enough information to make the connection. dialog options that were passed as valid parameters are dimmed and unavailable. The driver's login dialog is never displayed.

The **SQLOpen** function will never return an invalid connection ID. The following example establishes a connection using the driver's login dialog:

```
id& = SQLOpen("",,1)
```

The compiler returns 0 and generates a trappable runtime error if **SQLOpen** fails. Additional error information can then be retrieved using the **SQLError** function.

Before you can use any SQL statements, you must set up a data source and relate an existing database to it. This is accomplished using the odbcadm.exe program.

Example

```
Sub Main
  Dim s As String
  id& = SQLOpen("dsn=SAMPLE",s$,3)
  Session.Echo "The completed connection string is: " & s$
  id& = SQLClose(id&)
End Sub
```

See Also SQL Access on page 19

SQLRequest

Syntax **SQLRequest(*connectionstr*, *querytext*, *resultarray* [, [*outputref*] [, [*driverprompt*] [, *colnameslogical*]]])**

Description Opens a connection, runs a query, and returns the results as an array. The **SQLRequest** function takes the following named parameters:

Parameter	Description
connectionstr	String specifying the connection information required to connect to the data source.
Querytext	String specifying the query to execute. The syntax of this string must strictly follow the syntax of the ODBC driver.
Resultarray	Array of variants to be filled with the results of the query. The **resultarray** parameter must be dynamic: it will be resized to hold the exact number of records and fields.
Outputref	Optional string to receive the completed connection string as returned by the driver.
Driverprompt	Optional integer specifying the behavior of the driver's dialog.
Colnameslogical	Optional boolean specifying whether the column names are returned as the first row of results. The default is False.

There is a runtime error if **SQLRequest** fails. Additional error information can then be retrieved using the **SQLError** function.

The **SQLRequest** function performs one of the following actions, depending on the type of query being performed:

Type of Query	Action
SELECT	The **SQLRequest** function fills **resultarray** with the results of the query, returning a long containing the number of results placed in the array. The array is filled as follows (assuming an **x** by **y** query):

Type of Query	Action
	(record 1,field 1) **(record 1,field 2)** **:** **(record 1,field y)** **(record 2,field 1)** **(record 2,field 2)** **:** **(record 2,field y)** **:** **:** **(record x,field 1)** **(record x,field 2)** **:** (record **x**,field **y**)
INSERT, DELETE, UPDATE	The **SQLRequest** function erases **resultarray** and returns a long containing the number of affected rows.

Example

```
Sub Main
  Dim a() As Variant
  l& = SQLRequest("dsn=SAMPLE;","Select * From c:\sample.dbf",a,,3,True)
  For x = 0 To Ubound(a)
    For y = 0 To l - 1
      Session.Echo a(x,y)
    Next y
  Next x
End Sub
```

SQLRetrieve

Syntax **SQLRetrieve(*connectionnum*, *resultarray*[, [*maxcolumns*] [, [*maxrows*] [, [*colnameslogical*] [, *fetchfirstlogical*]]]])**

Description Retrieves the results of a query. This function is called after a connection to a data source is established, a query is executed, and the desired columns are bound. The following table describes the named parameters to the **SQLRetrieve** function:

Parameter	Description
connectionnum	Long identifying a valid connected data source with pending query results.
Resultarray	Two-dimensional array of variants to receive the results. The array has **x** rows by **y** columns. The number of columns is determined by the number of bindings on the connection.

Parameter	Description
`Maxcolumns`	Optional integer expression specifying the maximum number of columns to be returned. If **`maxcolumns`** is greater than the number of columns bound, the additional columns are set to empty. If **`maxcolumns`** is less than the number of bound results, the rightmost result columns are discarded until the result fits.
`Maxrows`	Optional integer specifying the maximum number of rows to be returned. If **`maxrows`** is greater than the number of rows available, all results are returned, and additional rows are set to empty. If **`maxrows`** is less than the number of rows available, the array is filled, and additional results are placed in memory for subsequent calls to SQLRetrieve.
`Colnameslogical`	Optional boolean specifying whether column names should be returned as the first row of results. The default is False.
`Fetchfirstlogical`	Optional boolean expression specifying whether results are retrieved from the beginning of the result set. The default is False. Before you can retrieve the results from a query, you must: Initiate a query by calling the **`SQLExecQuery`** function Specify the fields to retrieve by calling the **`SQLBind`** function.

This function returns a long specifying the number of rows available in the array.

There is a runtime error if SQLRetrieve fails. Additional error information is placed in memory.

Example

```
Sub Main
  Dim a() As Variant
  Dim b() As Variant
  Dim c() As Variant
  On Error Goto Trap
  id& = SQLOpen("DSN=SAMPLE",,3)
  qry& = SQLExecQuery(id&,"Select * From c:\sample.dbf"")
  i% = SQLBind(id&,b,3)
  i% = SQLBind(id&,b,1)
  i% = SQLBind(id&,b,2)
  i% = SQLBind(id&,b,6)
  l& = SQLRetrieve(id&,c)
  For x = 0 To Ubound(c,2)
    For y = 0 To l& - 1
      Session.Echo c(x,y)
    Next y
  Next x
  id& = SQLClose(id&)
  Exit Sub
Trap:
  rc% = SQLError(a)
  If (rc%) Then
    For x = 0 To (rc% - 1)
      Session.Echo "The SQLState returned was: " & a(x,0)
      Session.Echo "The native error code returned was: " & a(x,1)
      Session.Echo a(x,2)
```

```
    Next x
  End If
End Sub
```

See Also SQL Access on page 19

SQLRetrieveToFile

Syntax **SQLRetrieveToFile(*connectionnum*, *destination* [, [*colnameslogical*] [, *columndelimiter*]])**

Description Retrieves the results of a query and writes them to the specified file. The following table describes the named parameters to the **SQLRetrieveToFile** function:

Parameter	Description
connectionnum	Long specifying a valid connection ID.
Destination	String specifying the file where the results are written.
Colnameslogical	Optional boolean specifying whether the first row of results returned are the bound column names. By default, the column names are not returned.
Columndelimiter	Optional string specifying the column separator. A tab (**Chr$(9)**) is used as the default.

Before you can retrieve the results from a query, you must (1) initiate a query by calling the **SQLExecQuery** function and (2) specify the fields to retrieve by calling the **SQLBind** function.

This function returns the number of rows written to the file. A runtime error is generated if there are no pending results or if the compiler is unable to open the specified file.

There is a runtime error if **SQLRetrieveToFile** fails. Additional error information may be placed in memory for later use with the **SQLError** function.

Example

```
Sub Main
  Dim a() As Variant
  Dim b() As Variant
  On Error Goto Trap
  id& = SQLOpen("DSN=SAMPLE;UID=RICH",,4)
  t& = SQLExecQuery(id&, "Select * From c:\sample.dbf"")
  i% = SQLBind(id&,b,3)
  i% = SQLBind(id&,b,1)
  i% = SQLBind(id&,b,2)
  i% = SQLBind(id&,b,6)
  l& = SQLRetrieveToFile(id&,"c:\results.txt",True,",")
  id& = SQLClose(id&)
  Exit Sub
Trap:
  rc% = SQLError(a)
  If (rc%) Then
    For x = 0 To (rc-1)
      Session.Echo "The SQLState returned was: " & a(x,0)
```

```
      Session.Echo "The native error code returned was: " & a(x,1)
      Session.Echo a(x,2)
    Next x
  End If
End Sub
```

See Also SQL Access on page 19

Sqr

Syntax **Sqr(*number*)**

Description Returns a **Double** representing the square root of **number**. The **number** parameter is a **Double** greater than or equal to 0.

See Also **This example calculates the square root of the numbers from 1 to 10 and displays them.**

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  For x = 1 To 10
    sx# = Sqr(x)
    mesg = mesg & Format(x,"Fixed") & " - " & Format(sx#,"Fixed") & crlf
  Next x
  Session.Echo mesg
End Sub
```

Stop

Syntax **Stop**

Description Suspends execution of the current macro, returning control to the debugger.

Example

```
Sub Main
  For x = 1 To 10
    z = Random(0,10)
    If z = 0 Then Stop
    y = x / z
  Next x
End Sub
```

See Also Macro Control and Compilation on page 10

Str, Str$

Syntax **Str[$](number)**

Description Returns a string representation of the given number. The **number** parameter is any numeric expression or expression convertible to a number. If **number** is negative, then the returned string will contain a leading minus sign. If **number** is positive, then the returned string will contain a leading space.

Singles are printed using only 7 significant digits. Doubles are printed using 15–16 significant digits.

These functions only output the period as the decimal separator and do not output thousands separators. Use the CStr, Format, or Format$ function for this purpose.

Example

```
Sub Main
  x# = 100.22
  Session.Echo "The string value is: " + Str(x#)
End Sub
```

See Also Character and String Manipulation on page 3

StrComp

Syntax `StrComp(string1,string2 [,compare])`

Description Returns an **`Integer`** indicating the result of comparing the two string arguments. One of the following values is returned:

Value	Description
0	**`string1`** = **`string2`**
1	**`string1`** > **`string2`**
–1	**`string1`** < **`string2`**
Null	**`string1`** or **`string2`** is null

The **`StrComp`** function accepts the following parameters:

Parameter	Description
`string1`	First string to be compared, which can be any expression convertible to a string.
`string2`	Second string to be compared, which can be any expression convertible to a string.
`Compare`	Optional integer specifying how the comparison is to be performed. It can be either of the following values:
`0`	Case-sensitive comparison
`1`	Case-insensitive comparison
	If **`compare`** is not specified, then the current Option Compare setting is used. If no Option Compare statement has been encountered, then Binary is used (i.e., string comparison is case-sensitive).

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  a$ = "This string is UPPERCASE and lowercase"
  b$ = "This string is uppercase and lowercase"
```

```
  c$ = "This string"
  d$ = "This string is uppercase and lowercase characters"
  abc = StrComp(a$,b$,0)
  mesg = mesg & "a and c (sensitive)  : " & Format(abc,"True/False") & crlf
  abi = StrComp(a$,b$,1)
  mesg = mesg & "a and b (insensitive): " & Format(abi,"True/False") & crlf
  aci = StrComp(a$,c$,1)
  mesg = mesg & "a and c (insensitive): " & Format(aci,"True/False") & crlf
  bdi = StrComp(b$,d$,1)
  mesg = mesg & "b and d (sensitive)  : " & Format(bdi,"True/False") & crlf
  Session.Echo mesg
End Sub
```

See Also Character and String Manipulation on page 3; Keywords, Data Types, Operators, and Expressions on page 6

StrConv

Syntax `StrConv(string, conversion)`

Description Converts a string based on a conversion parameter. The StrConv function takes the following named parameters:

Parameter	Description
string	A string expression specifying the string to be converted.
Conversion	An integer specifying the types of conversions to be performed.

The **conversion** parameter can be any combination of the following constants:

Constant	Value	Description
ebUpperCase	1	Converts **string** to uppercase.
ebLowerCase	2	Converts **string** to lowercase.
ebProperCase	3	Capitalizes the first letter of each word.
ebWide	4	Converts narrow characters to wide characters. This constant is supported on Japanese locales only.
ebNarrow	8	Converts wide characters to narrow characters. This constant is supported on Japanese locales only.
ebKatakana	16	Converts Hiragana characters to Katakana characters. This constant is supported on Japanese locales only.

Constant	Value	Description
ebHiragana	32	Converts Katakana characters to Hiragana characters. This constant is supported on Japanese locales only.
ebUnicode	64	Converts string from MBCS to UNICODE. (This constant can only be used on Windows NT, which supports UNICODE.)
ebFromUnicode	128	Converts string from UNICODE to MBCS. (This constant can only be used on Windows NT, which supports UNICODE.)

A runtime error is generated when an unsupported conversion is requested. For example, the **ebWide** and **ebNarrow** constants can only be used on an MBCS platform.

The following groupings of constants are mutually exclusive and therefore cannot be specified at the same time:

```
ebUpperCase, ebLowerCase, ebProperCase
ebWide, ebNarrow
ebUnicode, ebFromUnicode
```

Many of the constants can be combined. For example, **ebLowerCase Or ebNarrow**.

When converting to proper case (i.e., the **ebProperCase** constant), the following are seen as word delimiters: tab, linefeed, carriage-return, formfeed, vertical tab, space, null.

Example

```
Sub Main
  a = InputBox("Type any string:")
  Session.Echo "Upper case: " & StrConv(a,ebUpperCase)
  Session.Echo "Lower case: " & StrConv(a,ebLowerCase)
  Session.Echo "Proper case: " & StrConv(a,ebProperCase)
End Sub
```

See Also Character and String Manipulation on page 3

String (data type)

Syntax **String**

Description Capable of holding a number of characters. Strings are used to hold sequences of characters, each character having a value between 0 and 255. Strings can be any length up to a maximum length of 32767 characters. Strings can contain embedded nulls, as shown in the following example:

```
s$ = "Hello" + Chr$(0) + "there"    'String with embedded null
```

The length of a string can be determined using the **Len** function. This function returns the number of characters that have been stored in the string, including unprintable characters.

The type-declaration character for string is **$**.

String variables that have not yet been assigned are set to zero-length by default.

Strings are normally declared as variable-length, meaning that the memory required for storage of the string depends on the size of its content. The following statements declare a variable-length string and assign it a value of length 5:

```
Dim s As String
s = "Hello"        'String has length 5.
```

Fixed-length strings are given a length in their declaration:

```
Dim s As String * 20
s = "Hello"        'String length = 20 with spaces to end of string.
```

When a string expression is assigned to a fixed-length string, the following rules apply:

- If the string expression is less than the length of the fixed-length string, then the fixed-length string is padded with spaces up to its declared length.
- If the string expression is greater than the length of the fixed-length string, then the string expression is truncated to the length of the fixed-length string.

Fixed-length strings are useful within structures when a fixed size is required, such as when passing structures to external routines.

The storage for a fixed-length string depends on where the string is declared, as described in the following table:

Declared	Stored
In structures	In the same data area as that of the structure. Local structures are on the stack; public structures are stored in the public data space; and private structures are stored in the private data space. Local structures should be used sparingly as stack space is limited.
In arrays	In the global string space along with all the other array elements.
In local routines	On the stack. The stack is limited in size, so local fixed-length strings should be used sparingly.

See Also Character and String Manipulation on page 3; Keywords, Data Types, Operators, and Expressions on page 6

String, String$

Syntax `String[$](number, character)`

Description Returns a string of length **`number`** consisting of a repetition of the specified filler character. **`String$`** returns a **`String`**, whereas **`String`** returns a **`String`** variant. These functions take the following named parameters:

Parameter	Description
`number`	Integer specifying the number of repetitions.
`Character`	Integer specifying the character code to be used as the filler character. If **`character`** is greater than 255 (the largest character value), then the compiler converts it to a valid character using the following formula: **`character Mod 256`**. If character is a string, then the first character of that string is used as the filler character.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  a$ = "This string will appear underlined."
  b$ = String$(Len(a$),"=")
  Session.Echo a$ & crlf & b$
End Sub
```

See Also Character and String Manipulation on page 3

Sub...End Sub

Syntax

```
[Private | Public] [Static] Sub name[(arglist)]
  [statements]
End Sub
```

where **`arglist`** is a comma-separated list of the following (up to 30 arguments are allowed):

```
[Optional] [ByVal | ByRef] parameter[()] [As type]
```

Note that a comment line must immediately follow the initial Sub line. This line is intended to identify who created the macro and when. The comment line format is:

'! **`Macro created by`** ***`name`*** **`on`** ***`date`*****`.`**

You must at least include a **`'!`** line.

Description Declares a subroutine. The **`Sub`** statement has the following parts:

Part	Description
Private	Indicates that the subroutine being defined cannot be called from other macros in other modules.
Public	Indicates that the subroutine being defined can be called from other macros in other modules. If the Private and Public keywords are both missing, then Public is assumed.
Static	Recognized by the compiler but currently has no effect.

Part	Description
`Name`	Name of the subroutine, which must follow naming conventions: Must start with a letter. May contain letters, digits, and the underscore character (_). Punctuation and type-declaration characters are not allowed. The exclamation point (!) can appear within the name as long as it is not the last character. Must not exceed 80 characters in length.
Optional	Keyword indicating that the parameter is optional. All optional parameters must be of type variant. Furthermore, all parameters that follow the first optional parameter must also be optional. If this keyword is omitted, then the parameter is required. **Note:** You can use the **`IsMissing`** function to determine whether an optional parameter was actually passed by the caller.
ByVal	Keyword indicating that the parameter is passed by value.
ByRef	Keyword indicating that the parameter is passed by reference. If neither the ByVal nor the ByRef keyword is given, then ByRef is assumed.
`Parameter`	Name of the parameter, which must follow the same naming conventions as those used by variables. This name can include a type-declaration character, appearing in place of **`As type`**.
`Type`	Type of the parameter (i.e., integer, string, and so on). Arrays are indicated with parentheses. For example, an array of integers is declared: **`Sub Test(a() As Integer)End Sub`**

A subroutine terminates when one of the following statements is encountered:

```
End Sub
Exit Sub
```

Subroutines can be recursive.

Passing Parameters to Subroutines

Parameters are passed to a subroutine either by value or by reference, depending on the declaration of that parameter in **`arglist`**. If the parameter is declared using the **`ByRef`** keyword, then any modifications to that passed parameter within the subroutine change the value of that variable in the caller. If the parameter is declared using the **`ByVal`** keyword, then the value of that variable cannot be changed in the called subroutine. If neither the **`ByRef`** nor the **`ByVal`** keyword is specified, then the parameter is passed by reference.

You can override passing a parameter by reference by enclosing that parameter within parentheses. For instance, the following example passes the variable j by reference, regardless of how the third parameter is declared in the **arglist** of **UserSub**:

```
UserSub 10,12,(j)
```

Optional Parameters

You can skip parameters when calling subroutines, as shown in the following example:

```
Sub Test(a%,b%,c%)
End Sub

Sub Main
  Test 1,,4      'Parameter 2 was skipped.
End Sub
```

You can skip any parameter with the following restrictions:

- The call cannot end with a comma. For instance, using the above example, the following is not valid:

```
  Test 1,,
```

The call must contain the minimum number of parameters as required by the called subroutine. For instance, using the above example, the following are invalid:

```
  Test ,1      'Only passes two out of three required parameters.
  Test 1,2      'Only passes two out of three required parameters.
```

When you skip a parameter in this manner, the compiler creates a temporary variable and passes this variable instead. The value of this temporary variable depends on the data type of the corresponding parameter in the argument list of the called subroutine, as described in the following table:

Value	Data Type
0	Integer, long, single, double, currency
Zero-length string	String
Nothing	Object (or any data object)
Error	Variant
December 30, 1899	Date
False	Boolean

Within the called subroutine, you will be unable to determine whether a parameter was skipped unless the parameter was declared as a variant in the argument list of the subroutine. In this case, you can use the **IsMissing** function to determine whether the parameter was skipped:

```
Sub Test(a,b,c)
  If IsMissing(a) Or IsMissing(b) Then Exit Sub
End Sub
```

Example

```
Sub Main
  r! = 10
  PrintArea r!
End Sub
Sub PrintArea(r as single)
  area! = (r! ^ 2) * Pi
  Session.Echo "The area of a circle with radius " & r! & " = " & area!
End Sub
```

See Also Macro Control and Compilation on page 10

Switch

Syntax `Switch(condition1,expression1 [,condition2,expression2 ... [,condition7,expression7]])`

Description Returns the expression corresponding to the first **True** condition. The **Switch** function evaluates each condition and expression, returning the expression that corresponds to the first condition (starting from the left) that evaluates to **True**. Up to seven condition/expression pairs can be specified. A runtime error is generated it there is an odd number of parameters (i.e., there is a condition without a corresponding expression). The Switch function returns null if no condition evaluates to **True**.

Example

```
wd = Weekday(date)
strwd = switch(wd=1, "Sunday", wd=2, "Monday", wd=3, "Tuesday",
  wd=4, "Wednesday", wd=5, "Thursday", _
  wd=6, "Friday", wd=7, "Saturday")
Session.Echo "Today is " & strwd
End Sub
```

See Also Macro Control and Compilation on page 10

SYD

Syntax `SYD(cost, salvage, life, period)`

Description Returns the sum of years' digits depreciation of an asset over a specific period of time. The **SYD** of an asset is found by taking an estimate of its useful life in years, assigning values to each year, and adding up all the numbers. The formula used to find the SYD of an asset is as follows:

```
(Cost – Salvage_Value) * Remaining_Useful_Life / SYD
```

The **SYD** function requires the following named parameters:

Parameter	Description
`cost`	Double representing the initial cost of the asset.
`Salvage`	Double representing the estimated value of the asset at the end of its useful life.
`Life`	Double representing the length of the asset's useful life.
`Period`	Double representing the period for which the depreciation is to be calculated. It cannot exceed the life of the asset.

To receive accurate results, the parameters **`life`** and **`period`** must be expressed in the same units. If **`life`** is expressed in terms of months, for example, then **`period`** must also be expressed in terms of months.

Example In this example, an asset that cost $1,000.00 is depreciated over ten years. The salvage value is $100.00, and the sum of the years' digits depreciation is shown for each year.

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  For x = 1 To 10
    dep# = SYD(1000,100,10,x)
    mesg = mesg & "Year: " & x & "  Dep: " & Format(dep#,"Currency") & crlf
  Next x
  Session.Echo mesg
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

T

Tab

Syntax `Tab (column)`

Description Prints the number of spaces necessary to reach a given column position.

Note This function can only be used with the **`Print`** and **`Print#`** statements.

The **`column`** parameter is an **`Integer`** specifying the desired column position to which to advance. It can be any value between 0 and 32767 inclusive.

Rule 1: If the current print position is less than or equal to **`column`**, then the number of spaces is calculated as:

```
column – print_position
```

Rule 2: If the current print position is greater than **`column`**, then **`column`** – 1 spaces are printed on the next line.

If a line width is specified (using the **`Width`** statement), then the column position is adjusted as follows before applying the above two rules:

```
column = column Mod width
```

The **`Tab`** function is useful for making sure that output begins at a given column position, regardless of the length of the data already printed on that line.

Example

```
Sub Main
  Viewport.Open
  Print "Column1";Tab(10);"Column2";Tab(20);"Column3"
  Print Tab(3);"1";Tab(14);"2";Tab(24);"3"
  Sleep(10000)              'Wait 10 seconds.
  Viewport.Close
End Sub
```

See Also Drive, Folder, and File Access on page 4

Tan

Syntax `Tan(number)`

Description Returns a `Double` representing the tangent of `number.` The `number` parameter is a `Double` value given in radians.

Example

```
Sub Main
  c# = Tan(Pi / 4)
  Session.Echo "The tangent of 45 degrees is: " & c#
End Sub
```

See Also Numeric, Math, and Accounting Functions on page 9

Text

Syntax `Text x,y,width,height,title$ [,[.Identifier] [,[FontName$] [,[size] [,style]]]]`

Description Defines a text control within a dialog template. The text control only displays text; the user cannot set the focus to a text control or otherwise interact with it. The text within a text control word-wraps. Text controls can be used to display up to 32K of text. The `Text` statement accepts the following parameters:

Parameter	Description
`x`, `y`	Integer positions of the control (in dialog units) relative to the upper left corner of the dialog.
`width`, `height`	Integer dimensions of the control in dialog units.
`title$`	String containing the text that appears within the text control. This text may contain an ampersand character to denote an accelerator letter, such as "&Save" for Save. Pressing this accelerator letter sets the focus to the control following the Text statement in the dialog template.
`.Identifier`	Name by which this control can be referenced by statements in a dialog function (such as `DlgFocus` and `DlgEnable`). If this parameter is omitted, then the first two words from `title$` are used.
`FontName$`	Name of the font used for display of the text within the text control. If this parameter is omitted, then the default font for the dialog is used.
`size`	Size of the font used for display of the text within the text control. If this parameter is omitted, then the default size for the default font of the dialog is used.
`style`	Style of the font used for display of the text within the text control. This can be any of the following values: `ebRegular` Normal font (i.e., neither bold nor italic)

Parameter	Description	
	ebBold	Bold font
	ebItalic	Italic font
	ebBoldItalic	Bold-italic font. If this parameter is omitted, then **ebRegular** is used.

Accelerators are underlined, and the Alt+letter accelerator combination is used.

Example

```
Begin Dialog UserDialog3 81,64,128,60,"Untitled"
  CancelButton 80,32,40,14
  OKButton 80,8,40,14
  Text 4,8,68,44,"This text is displayed in the dialog."
End Dialog
```

See Also User Interaction on page 16

TextBox

Syntax **TextBox x,y,width,height,.Identifier [,[isMultiline] [,[FontName$] [,[size] [,style]]]]**

Description Defines a single or multiline text-entry field within a dialog template. The **TextBox** statement requires the following parameters:

Parameter	Description
x, **y**	Integer position of the control (in dialog units) relative to the upper left corner of the dialog.
width, **height**	Integer dimensions of the control in dialog units.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). This parameter also creates a string variable whose value corresponds to the content of the text box. This variable can be accessed using the syntax **DialogVariable.Identifier**
isMultiline	Specifies whether the text box can contain more than a single line (0 = single-line; 1 = multiline).
FontName$	Name of the font used for display of the text within the text box control. If this parameter is omitted, then the default font for the dialog is used.
size	Size of the font used for display of the text within the text box control. If this parameter is omitted, then the default size for the default font of the dialog is used.
style	Style of the font used for display of the text within the text box control. This can be any of the following values:

Parameter	Description	
	`ebRegular`	Normal font (i.e., neither bold nor italic)
	`ebBold`	Bold font
	`ebItalic`	Italic font
	`ebBoldItalic`	Bold-italic font. If this parameter is omitted, then **`ebRegular`** is used.

`If isMultiline` is 1, the **`TextBox`** statement creates a multiline text-entry field. When the user types into a multiline field, pressing the Enter key creates a new line rather than selecting the default button.

The **`isMultiLine`** parameter also specifies whether the text box is read-only and whether the text-box should hide input for password entry. To specify these extra parameters, you can form the **`isMultiLine`** parameter by ORing together the following values:

Value	Meaning
`0`	Text box is single-line.
`1`	Text box is multi-line.
`&H8000`	Text box is read-only.
`&H4000`	Text box is password-entry.

For example, the following statement creates a read-only multiline text box:

```
TextBox 10,10,80,14,.TextBox1,1 Or &H8000
```

The **`TextBox`** statement can only appear within a dialog template (i.e., between the **`Begin Dialog`** and **`End Dialog`** statements).

When the dialog is created, the **`.Identifier`** variable is used to set the initial content of the text box. When the dialog is dismissed, the variable will contain the new content of the text box.

A single-line text box can contain up to 256 characters. The length of text in a multiline text box is the default memory limit specified by Windows 98/Me.

Example

```
Begin Dialog UserDialog3 81,64,128,60,"Untitled"
  CancelButton 80,32,40,14
  OKButton 80,8,40,14
  TextBox 4,8,68,44,.TextBox1,1
End Dialog
```

See Also User Interaction on page 16

Time, Time$ (functions)

Syntax `Time[$][()]`

Description Returns the system time as a **`String`** or as a **`Date`** variant. The **`Time$`** function returns a string that contains the time in a 24-hour time format, whereas **`Time`** returns a **`Date`** variant. To set the time, use the **`Time/Time$`** statements.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  oldtime$ = Time$
  mesg = "Time was: " & oldtime$ & crlf
  Time$ = "10:30:54"
  mesg = mesg & "Time set to: " & Time$ & crlf
  Time$ = oldtime$
  mesg = mesg & "Time restored to: " & Time$
  Session.Echo mesg
End Sub
```

See Also Time and Date Access on page 17

Time, Time$ (statements)

Syntax `Time[$] = newtime`

Description Sets the system time to the time contained in the specified string. The **`Time$`** statement requires a string variable in one of the following formats:

```
HH
HH:MM
HH:MM:SS
```

where **`HH`** is between 0 and 23, **`MM`** is between 0 and 59, and **`SS`** is between 0 and 59.

The **`Time`** statement converts any valid expression to a time, including string and numeric values. Unlike the **`Time$`** statement, **`Time`** recognizes many different time formats, including 12-hour times.

Note You may not have permission to change the time, causing runtime error 70 to be generated.

Example

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  oldtime$ = Time$
  mesg = "Time was: " & oldtime$ & crlf
  Time$ = "10:30:54"
  mesg = mesg & "Time set to: " & Time$ & crlf
  Time$ = oldtime$
  mesg = mesg & "Time restored to: " & Time$
  Session.Echo mesg
End Sub
```

See Also Time and Date Access on page 17

Timer

Syntax **`Timer`**

Description Returns a **`Single`** representing the number of seconds that have elapsed since midnight.

Example

```
Sub Main
  start& = Timer
  Session.Echo "Click the OK button, please."
  total& = Timer - start&
  Session.Echo "The elapsed time was: " & total& & " seconds."
End Sub
```

See Also Time and Date Access on page 17

TimeSerial

Syntax **`TimeSerial(hour, minute, second)`**

Description Returns a **`Date`** variant representing the given time with a date of zero. The **`TimeSerial`** function requires the following named parameters:

Parameter	Description
`hour`	Integer between 0 and 23.
`Minute`	Integer between 0 and 59.
`Second`	Integer between 0 and 59.

Example

```
Sub Main
  start# = TimeSerial(10,22,30)
  finish# = TimeSerial(10,35,27)
  dif# = Abs(start# - finish#)
  Session.Echo "The time difference is: " & Format(dif#, "hh:mm:ss")
End Sub
```

See Also Time and Date Access on page 17

TimeValue

Syntax **`TimeValue(time)`**

Description Returns a **`Date`** variant representing the time contained in the specified string argument. This function interprets the passed **`time`** parameter looking for a valid time specification. The **`time`** parameter can contain valid time items separated by time separators such as colon (**`:`**) or period (**`.`**). Time strings can contain an optional date specification, but this is not used in the formation of the returned value. If a particular time item is missing, then it is set to 0. For example, the string "10 pm" would be interpreted as "22:00:00."

Example

```
Sub Main
  t1$ = "10:15"
  t2# = TimeValue(t1$)
  Session.Echo "The TimeValue of " & t1$ & " is: " & t2#
End Sub
```

See Also Time and Date Access on page 17

Transfer (object)

The **`Transfer`** object is the current transfer method in use by the active session. With the **`Transfer`** object you control or have access to those properties of SmarTerm that relate to file transfer, such as generic File menu commands and any settings that appear on the Properties>File Transfer Properties dialog (which vary depending on the transfer method). You can also access methods that relate to the details of host connection (which also vary depending on the transfer method).

Note For macro commands dealing with data capture from the host, see the methods and properties of the Session object.

All methods and properties unique to a given transfer method are prefixed with the name of the transfer method, such as Transfer.FTPHostName. As of this version of SmarTerm, the supported file transfer methods are FTP, IND$FILE, Kermit, XModem, YModem, and ZModem. However, because ZModem handles so many file transfer issues automatically, there are no unique Transfer properties or methods for it.

Transfer.Command

Kermit and FTP file transfer protocols only

Syntax **`Transfer.Command(commandtext$)`**

where **`commandtext$`** is the command to execute (string).

Description Allows commands to be sent to the current SmarTerm file transfer method, returning the command’s completion status (Boolean).

Example

```
Sub Main
  Dim RetVal as Boolean
  RetVal = Transfer.Command("cwd /pub/samples")
  If RetVal = False Then
     GoTo ErrorHandler
  End If
  RetVal = Transfer.Command("lcd c:\incoming")
  If RetVal = False Then
     GoTo ErrorHandler
  End If
  RetVal = Transfer.Command("mget file1 file2")
  If RetVal = False Then
     GoTo ErrorHandler
  End If
  End
```

```
  ErrorHandler:
  Session.Echo "An error occurred, stopping the macro."
  End
End Sub
```

See Also File Transfer on page 2

Transfer.FTPAutoConnect

Syntax `Transfer.FTPAutoConnect`

Description Returns or sets whether an FTP connection should be established automatically (boolean).

Example

```
Sub Main
  Dim AutoConnect as Boolean
  AutoConnect = Transfer.FTPAutoConnect
  Transfer.FTPAutoConnect = True
End Sub
```

See Also File Transfer on page 2

Transfer.FTPConfirmDeleteFiles

Syntax `Transfer.FTPConfirmDeleteFiles`

Description Returns or sets whether or not FTP will display a dialog confirming the potential deletion of a file (Boolean). If set to TRUE (the default), and the macro detects that a file will be deleted, then the macro pauses until the user responds to the confirmation dialog. If set to FALSE, then the macro deletes the file without confirmation.

Note There must be an active FTP connection for this property to take effect; you cannot set this property and then make the FTP connection. This is demonstrated in the example.

Example

```
'This example deletes files via FTP without confirmation
'It assumes an open connection, but tests anyway.
Sub Main

If Transfer.Command("dir") = TRUE Then
   Transfer.FTPConfirmDeleteFiles = FALSE
   MsgBox "File will be deleted without warning!"
   Transfer.Command("mdel *.*")
Else
   MsgBox "Not connected. Exiting macro."
End If

End Sub
```

See Also File Transfer on page 2

Transfer.FTPConfirmRemoveFolders

Syntax `Transfer.FTPConfirmRemoveFolders`

Description Returns or sets whether or not FTP will display a dialog confirming the potential removal of a folder (Boolean). If set to TRUE (the default), and the macro detects that a folder will be removed, then the macro pauses until the user responds to the confirmation dialog. If set to FALSE, then the macro removes the folder without confirmation.

Note There must be an active FTP connection for this property to take effect; you cannot set this property and then make the FTP connection. This is demonstrated in the example.

Example

```
'This example removes folders via FTP without confirmation
'It assumes an open connection, but tests anyway.
Sub Main

If Transfer.Command("dir") = TRUE Then
   Transfer.FTPConfirmRemoveFolders = FALSE
   MsgBox "Folders will be removed without warning!"
   Transfer.Command("rmdir .")
Else
   MsgBox "Not connected. Exiting macro."
End If

End Sub
```

See Also File Transfer on page 2

Transfer.FTPConfirmReplaceFiles

Syntax `Transfer.FTPConfirmReplaceFiles`

Description Returns or sets whether or not FTP will display a dialog confirming the potential replacement of a file (Boolean). If set to TRUE (the default), and the macro detects that a file will be replaced, then the macro pauses until the user responds to the confirmation dialog. If set to FALSE, then the macro replaces the file without confirmation.

Note There must be an active FTP connection for this property to take effect; you cannot set this property and then make the FTP connection. This is demonstrated in the example.

Example

```
'This example replaces files via FTP without confirmation
'It assumes an open connection, but tests anyway.
Sub Main

|If Transfer.Command("dir") = TRUE Then
   Transfer.FTPConfirmReplaceFiles = FALSE
   MsgBox "File will be replaced without warning!"
   Transfer.Command("mget *.*")
Else
   MsgBox "Not connected. Exiting macro."
End If

End Sub
```

See Also File Transfer on page 2

Transfer.FTPConfirmTransferFiles

Syntax `Transfer.FTPConfirmTransferFiles`

Description Returns or sets whether or not FTP will display a dialog confirming file transfer (Boolean). If set to TRUE, and the macro detects that a file will be transfered, then the macro pauses until the user responds to the confirmation dialog. If set to FALSE (the default), then the macro transfers the file without confirmation.

Note There must be an active FTP connection for this property to take effect; you cannot set this property and then make the FTP connection. This is demonstrated in the example.

Example

```
'This example transfers files via FTP without confirmation
'It assumes an open connection, but tests anyway.
Sub Main

If Transfer.Command("dir") = TRUE Then
   Transfer.FTPConfirmTransferFiles = FALSE
   MsgBox "File will be transfered without warning!"
   Transfer.Command("mput *.*")
Else
   MsgBox "Not connected. Exiting macro."
End If

End Sub
```

See Also File Transfer on page 2

Transfer.FTPConfirmTransferFolders

Syntax Transfer.FTPConfirmTransferFolders

Description Returns or sets whether or not FTP will display a dialog confirming folder transfer (Boolean).

Note This property is included in support of future capabilities. FTP is not currently able to transfer folders.

See Also File Transfer on page 2

Transfer.FTPDeleteIncompleteFiles

Syntax Transfer.FTPDeleteIncompleteFiles

Description Returns or sets whether or not FTP will delete incomplete files (boolean). If set to true (default), the macro will tell ftp to delete incomplete files. If set to false, then FTP will not delete incomplete files.

See Also File Transfer on page 2

Example

```
Sub Main
```

```
'! This example downloads a file from a remote host using FTP
    Transfer.FTPHostName = "ftp.host.com"
    Transfer.FTPUserName = "User"
    Transfer.FTPUserPassword = "Password"
    Transfer.Command "Lcd 'c:\'"
    Transfer.Command "Type binary"
    Transfer.FTPDeleteIncompleteFiles=False
    Transfer.Command "Get SomeFile.dat"
    Transfer.Command "Quit"
End Sub
```

Transfer.FTPHostName

Telnet sessions only

Syntax `Transfer.FTPHostName`

Description Returns or sets the FTP host name (string).

Example

```
Sub Main
  Dim HostName as String
  HostName = Transfer.FTPHostName
  If HostName <> "ftp.host.com" Then
     Session.Echo "Using the ftp.host.com FTP site"
     Transfer.FTPHostName = "ftp.host.com"
  End If
End Sub
```

See Also File Transfer on page 2

Transfer.FTPUserName

Telnet sessions only

Syntax `Transfer.FTPUserName`

Description Returns or sets the FTP user name (string).

Example

```
Sub Main
  Dim UserName as String
  UserName = Transfer.FTPUserName
  If UserName <> "anonymous" Then
     Session.Echo "Using an anonymous login for this host."
     Transfer.FTPUserName = "anonymous"
  End If
End Sub
```

See Also File Transfer on page 2

Transfer.FTPUserPassword

Telnet sessions only

Syntax `Transfer.FTPUserPassword`

Description Returns or sets the FTP user password (string).

Example

```
Sub Main
  Dim Password as String
  Password = Transfer.FTPUserPassword
  If Password = "" Then
     Transfer.FTPUserPassword = "jarngy49"
  End If
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILEAdditionalCommands

3270 and 5250 sessions only

Syntax `Transfer.INDFILEAdditionalCommands`

Description Returns or sets the additional syntax to be added to a given IND$FILE command (string).

Example

```
Sub Main
  Dim Commands as string
  Commands = Transfer.INDFILEAdditionalCommands
  Transfer.INDFILEAdditionalCommands = “Quiet”
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILEEnableCRLFHandling

3270 and 5250 sessions only

Syntax `Transfer.INDFILEEnableCRLFHandling`

Description Returns or sets the CRLF (carriage return / line feed) processing for the selected file format (boolean). Possible values:

Value	Definition
True	Strip CRLF from each line of a file sent to the host, and add CRLF to each line received from the host.
False	Use the default processing for the selected file format.

Example

```
Sub Main
  Dim CRLF as boolean
  CRLF = Transfer.INDFILEEnableCRLFHandling
  Transfer.INDFILEEnableCRLFHandling = True
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILEHostEnvironment

3270 and 5250 sessions only

Syntax `Transfer.INDFILEHostEnvironment`

Description Returns or sets the host system environment (string). Possible values are:

Value	Definition
CICS	MVS/CICS
CMS	VM/CMS
TSO	MVS/TSO

Example

```
Sub Main
  Dim HostEnv as string
  HostEnv = Transfer.INDFILEHostEnvironment
  Transfer.INDFILEHostEnvironment = "CICS"
  MsgBox "The Previous Host Environment was: " & HostEnv
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILELocalFileFormat

3270 and 5250 sessions only

Syntax `Transfer.INDFILELocalFileFormat`

Description Returns or sets the format of the local file (string). Possible values:

Value	Definition
ASCII	Character translation is based on the current local system language. ASCII is the DOS standard format.
ANSI	Character translation is based on the character set selected in your session. ANSI is the Windows standard format.
Binary	The transfer takes place without character translation.

This property is supported where an extended terminal type is in use.

Example

```
Sub Main
  Dim FileFormat as string
  FileFormat = Transfer.INDFILELocalFileFormat
  Transfer.INDFILELocalFileFormat = "Binary"
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILELogicalRecordLength

3270 and 5250 sessions only

Syntax `Transfer.INDFILELogicalRecordLength`

Description Returns or sets the length of the set of data considered to be a logical record (integer). This number can be between 0 and 32761.

Example

```
Sub Main
  Dim LogicalRecordLength as integer
  LogicalRecordLength = Transfer.INDFILELogicalRecordLength
  Transfer.INDFILELogicalRecordLength = 255
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILEPacketSize

3270 and 5250 sessions only

Syntax `Transfer.INDFILEPacketSize`

Description Returns or sets the IND$FILE packet-size setting (integer). The default is 8Kb, which most hosts support; the number can be from 1 to 32Kb. Larger packet size means faster transfer. However, if you specify a value larger than your host supports, your session will be disconnected. This property is supported with extended mode terminal types.

Example

```
Sub Main
  Dim PktSize as integer
  PktSize = Transfer.INDFILEPacketSize
  Transfer.INDFILEPacketSize = 16
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILEPromptBeforeOverwrite

3270 and 5250 sessions only

Syntax `Transfer.INDFILEPromptBeforeOverwrite`

Description Returns or sets whether the user sees a prompt before a host-to-local transfer overwrites any existing files of the same name (boolean). Possible values:

Value	Definition
True	Prompt before overwriting existing files.
False	Overwrite without prompting.

Example

```
Sub Main
  Dim Prompt as boolean
  Prompt = Transfer.INDFILEPromptBeforeOverwrite
  Transfer.INDFILEPromptBeforeOverwrite = True
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILERecordFormat

3270 and 5250 sessions only

Syntax `Transfer.INDFILERecordFormat`

Description Returns or sets the record format of the file on the host (string). Possible values:

Value	Definition
Default	Accepts the host file's record format.
Fixed	Specifies that all records in the host file are the same length.
Undefined	Accepts that the host file's records are of undefined or unknown length.
Variable	Specifies that records in the host file can be of different lengths.

Example

```
Sub Main
  Dim RecordFormat as string
  RecordFormat = Transfer.INDFILERecordFormat
  Transfer.INDFILERecordFormat = "Variable"
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILEResponseTimeout

3270 and 5250 sessions only

Syntax `Transfer.INDFILEResponseTimeout`

Description Returns or sets the amount of time SmarTerm should wait for the host to respond to each IND$FILE command sent. The timeout range is 10 to 600 seconds; the default is 40 seconds (integer).

Example

```
Sub Main
  Dim Response as integer
  Response = Transfer.INDFILEResponseTimeout
  Transfer.INDFILEResponseTimeout = 20
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILEStartupTimeout

3270 and 5250 sessions only

Syntax `Transfer.INDFILEStartupTimeout`

Description Returns or sets the amount of time SmarTerm should wait for an initial response from the host before a startup attempt fails. The timeout range is 10 to 600 seconds; the default is 40 seconds (integer).

Example

```
Sub Main
  Dim Startup as integer
  Startup = Transfer.INDFILEStartupTimeout
  Transfer.INDFILEStartupTimeout = 20
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILETSOAllocationUnits

3270 and 5250 sessions only

Syntax `Transfer.INDFILETSOAllocationUnits`

Description Returns or sets the unit in which space is to be allocated (string). Possible values are:

Value	Definition
Blocks	Subdivision of a track.
Tracks	Path associated with a single read/write head as the data medium moves past it.
Cylinders	Set of all tracks that can be accessed without repositioning the access mechanism.
None	not in use

This property is supported in the TSO host environment only.

Example

```
Sub Main
  Dim Allocation as string
  Allocation = Transfer.INDFILETSOAllocationUnits
  Transfer.INDFILETSOAllocationUnits = "Blocks"
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILETSOAUPrimary

3270 and 5250 sessions only

Syntax `Transfer.INDFILETSOAUPrimary`

Description Returns or sets the number of units to be allocated (integer). The unit is defined in `Transfer.INDFILETSOAllocationUnits`.

This property is supported in the TSO host environment only.

Example

```
Sub Main
  Dim AUPrimary as integer
  AUPrimary = Transfer.INDFILETSOAUPrimary
  Transfer.INDFILETSOAUPrimary = 2000
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILETSOAUSecondary

3270 and 5250 sessions only

Syntax `Transfer.INDFILETSOAUSecondary`

Description Returns or sets the number of units to be allocated if the Primary number of units is exceeded (integer). The unit is defined in `Transfer.INDFILETSOAllocationUnits`.

This property is supported in the TSO host environment only.

Example

```
Sub Main
  Dim AUSecondary as integer
  AUSecondary = Transfer.INDFILETSOAUSecondary
  Transfer.INDFILETSOAUSecondary = 15
End Sub
```

See Also File Transfer on page 2

Transfer.INDFILETSOAverageBlockSize

3270 and 5250 sessions only

Syntax `Transfer.INDFILETSOAverageBlockSize`

Description Returns or sets the size of an average block, rather than having the host determine it (integer). Relevant only when Allocation Units is set to Block. Possible values are between 0 and 32760.

This property is supported in the TSO host environment only. It applies to all file formats.

Example

```
Sub Main
  Dim AvBlock as integer
  AvBlock = Transfer.INDFILETSOAverageBlockSize
  TRANSFER.INDFILETSOAverageBlockSize = 6200
 End Sub
```

See Also File Transfer on page 2

Transfer.INDFILETSOBlockSize

3270 and 5250 sessions only

Syntax `Transfer.INDFILETSOBlockSize`

Description Returns or sets the number of bytes to be allocated per block. This number can be between 0 and 32760. For fixed records, block size must be an even multiple of the logical record length. For variable records, block size must be equal to or greater than the largest record, plus 8 (integer).

This property is supported in the TSO host environment only.

Example

```
Sub Main
  Dim BlockSize as integer
  BlockSize = Transfer.INDFILETSOBlockSize
  Transfer.INDFILETSOBlockSize = 6160
End Sub
```

See Also File Transfer on page 2

Transfer.KermitCheckSumType

VT, ANSI, SCO, and DG sessions only

Syntax `Transfer.KermitCheckSumType`

Description Returns or sets the Kermit checksum-type setting. Possible values are:

```
"onebyte"
"twobyte"
"threebytecrc"
```

Example

```
Sub Main
  Dim CheckSum as String
  CheckSum = Transfer.KermitCheckSumType
  Transfer.KermitCheckSumType = "threebytecrc"
End Sub
```

See Also File Transfer on page 2

Transfer.KermitDuplicateFileWarning

VT, ANSI, SCO, and DG sessions only

Syntax `Transfer.KermitDuplicateFileWarning`

Description Returns or sets the Kermit duplicate-file-warning state (boolean).

Example

```
Sub Main
  Dim DupWarn as Boolean
  DupWarn = Transfer.KermitDuplicateFileWarning
  Transfer.KermitDuplicateFileWarning = True
End Sub
```

See Also File Transfer on page 2

Transfer.KermitPacketSize

VT, ANSI, SCO, and DG sessions only

Syntax `Transfer.KermitReceivePacketSize`

Description Returns or sets the Kermit send and receive packet-size setting (integer). Possible values for this property are: 94, 1024, 2048, 3072, 4096, 5120, 6144, 7168, 8192.

Example

```
Sub Main
  Dim PktSize as Integer
  PktSize = Transfer.KermitPacketSize
  Transfer.KermitPacketSize = 1024
End Sub
```

See Also File Transfer on page 2

Transfer.ProtocolName

Syntax `Transfer.ProtocolName`

Description Returns the name of the current file transfer protocol (string). `Transfer.ProtocolName` returns one of the following values:

```
XMODEM
YMODEM
ZMODEM
KERMIT
FTP
IND$FILE
```

Example

```
Sub Main
  Dim XferName as String
  XferName = Transfer.ProtocolName
  Session.Echo "The current file transfer protocol is " & XferName
End Sub
```

See Also File Transfer on page 2

Transfer.ReceiveFile

Syntax `Transfer.ReceiveFile(pcfilename$)`

where `pcfilename$` is the name of the file on the PC (string).

Description Invokes a receive file transfer in the active SmarTerm session, returning the command’s completion status (boolean).

Example

```
Sub Main
  Dim RetVal as Boolean
 'Change protocol to Kermit
  RetVal = Session.TransferProtocol("KERMIT")
  If RetVal = FALSE Then
```

```
      Goto ErrorHandler
   End IF

   'Start Transfer
   Session.Send "kermit" & Chr$(13)
   Session.Send "send filename.txt" & Chr$(13)
   sleep 2
   RetVal = Transfer.ReceiveFile("filename.txt")
   If RetVal = False Then
     Goto ErrorHandler
    End If
   End
   ErrorHandler:
     Session.Echo "The file transfer failed."
   End
End Sub
```

See Also File Transfer on page 2

Transfer.ReceiveFileAs

Syntax **`Transfer.ReceiveFileAs(hostfilename, pcfilename)`**

`Hostfilename` is the name of the file on the host and **`Pcfilename`** is the name of the file after transfer to the PC.

Description Invokes a receive file transfer in the active SmarTerm session, returning the completion status of the file transfer (boolean).

Example

```
'This example downloads a file to a PC using IND$FILE
Sub Main
'!
  Dim RetVal as Boolean
 'Change protocol to IND$FILE
  RetVal = Session.TransferProtocol("IND$FILE")
  If RetVal = FALSE Then
     Goto ErrorHandler
  End IF


  'Start Transfer
  RetVal = Transfer.ReceiveFileAs("hostexec.bak", "c:\autoexec.bat")
  If RetVal = False Then
    Goto ErrorHandler
   End If
  End
  ErrorHandler:
    msgbox "The file transfer failed."
  End
End Sub
```

See Also File Transfer on page 2

Transfer.SendFile

Syntax **`Transfer.SendFile(pcfilename$)`**

where **pcfilename$** is the name of the file on the PC (string).

Description Invokes a send file transfer, returning the completion status of the file transfer (boolean).

Example

```
Sub Main
  Dim RetVal as Boolean
 'Change protocol to YMODEM
  RetVal = Session.TransferProtocol("YMODEM")
  If RetVal = FALSE Then
     Goto ErrorHandler
  End IF

  'Start Transfer
  Session.Send "rb" & Chr$(13)
  sleep 2
  RetVal = Transfer.SendFile("c:\autoexec.bat")
  If RetVal = False Then
    Goto ErrorHandler
   End If
  End
  ErrorHandler:
    Session.Echo "The file transfer failed."
  End
End Sub
```

See Also File Transfer on page 2

Transfer.SendFileAs

Syntax **Transfer.SendFileAs(pcfilename, hostfilename)**

Pcfilename is the name of the file on the PC and **hostfilename** is the name of the file after transfer to the host.

To receive a file from the host, replace the send syntax in the example below with the receive syntax from above.

Description Invokes a send file transfer in the active SmarTerm session, returning the completion status of the file transfer (boolean).

Example

```
'This example uploads a file to a host using IND$FILE
Sub Main
'!
  Dim RetVal as Boolean
 'Change protocol to IND$FILE
  RetVal = Session.TransferProtocol("IND$FILE")
  If RetVal = FALSE Then
     Goto ErrorHandler
  End IF


  'Start Transfer
  Session.Send "rb" & chr$(13)
  sleep 2
```

```
  RetVal = Transfer.SendFileAs("c:\autoexec.bat", "hostexec.bak")
  If RetVal = False Then
    Goto ErrorHandler
   End If
  End
  ErrorHandler:
    msgbox "The file transfer failed."
  End
End Sub
```

See Also File Transfer on page 2

Transfer.Setup

Syntax `Transfer.Setup setupstring$`

where **`setupstring$`** is the string containing the setup specifications (string).

Description Sets file transfer parameters in SmarTerm.

Note This method is provided primarily for the support of PSL scripts.

The syntax of the string expression is identical between file transfer methods, although meaning varies somewhat. Specify setup options one at a time with their own **`Transfer.Setup`** statements, or more than one at a time, if you keep all options and settings within the quotation marks, separating the setup statements with commas:

```
Transfer.Setup "streaming = yes,checksumtype = crc16,packetsize = 128"
```

FTP transfers

```
Host name
HostName=  legal FTP host name or IP address
Transfer.Setup "hostname = unixbox"
User name
UserName=  legal FTP user name
Transfer.Setup "username = jpenn"
Password
UserPassword=  legal FTP password
Transfer.Setup "userpassword = mahler8"
Autoconnect
Autoconnect=  1
Autoconnect=  0
Transfer.Setup "autoconnect = 1"
```

KERMIT transfers

```
Discard partial file
DiscardPartialFile=  YES | NO
Transfer.Setup "discardpartialfile = yes"
Duplicate file warning
DuplicateFileWarning=  YES | NO
Transfer.Setup "duplicatefilewarning = yes"
Checksum type
ChecksumType=  OneByte | TwoByte | ThreeByteCRC
```

```
Transfer.Setup "checksumtype = threebytecrc"
Send packet size
SendPacketSize=  94 | 1024 | 2048 | 3072 | 4096 | 5120 | 6144 | 7168 | 8192
TRANSFER SETUP "sendpacketsize = 64"
Receive packet size
ReceivePacketSize=  94 | 1024 | 2048 | 3072 | 4096 | 5120 | 6144 | 7168 | 8192
TRANSFER SETUP "receivepacketsize = 512"
```

XMODEM, YMODEM, and ZMODEM transfers

```
Packet size
PacketSize=  128 | 1024
Transfer.Setup "packetsize = 128"
Checksum type
ChecksumType=  SIMPLE | CRC16
Transfer.Setup "checksumtype = crc16"
Streaming
Streaming=  YES | NO
Transfer.Setup "streaming = no"
```

See Also File Transfer on page 2

Transfer.XMODEMCheckSumType

VT, ANSI, SCO, and DG sessions only

Syntax `Transfer.XMODEMCheckSumType`

Description Returns or sets the XMODEM-checksum-type setting (string). **`Transfer.XMODEMCheckSumType`** accepts or returns one of the following strings: **`"simple"`** or **`"crc16"`**.

Example

```
Sub Main
  Dim CheckSum as String
  CheckSum = Transfer.XMODEMCheckSumType
  Transfer.XMODEMCheckSumType = "crc16"
End Sub
```

See Also File Transfer on page 2

Transfer.XMODEMPacketSize

VT, ANSI, SCO, and DG sessions only

Syntax `Transfer.XMODEMPacketSize`

Description Returns or sets the XMODEM-packet-size setting (integer). **`Transfer.XMODEMPacketSize`** accepts or returns either **`128`** or **`1024`**.

Example

```
Sub Main
  Dim PktSize as Integer
  PktSize = Transfer.XMODEMPacketSize
  Transfer.XMODEMPacketSize = 1024
End Sub
```

See Also File Transfer on page 2

Transfer.XMODEMStreaming

VT, ANSI, SCO, and DG sessions only

Syntax **`Transfer.XMODEMStreaming`**

Description Returns or sets a the XMODEM-streaming-mode setting (boolean).

Example

```
Sub Main
  Dim Streaming as Boolean
  Streaming = Transfer.XMODEMStreaming
  Transfer.XMODEMStreaming = False
End Sub
```

See Also File Transfer on page 2

Transfer.YMODEMCheckSumType

VT, ANSI, SCO, and DG sessions only

Syntax **`Transfer.YMODEMCheckSumType`**

Description Returns or sets the YMODEM-checksum-type setting (string). **`Transfer.YMODEMCheckSumType`** accepts or returns one of the following strings: **`"simple"`** or **`"crc16".`**

Example

```
Sub Main
  Dim CheckSum as String
  CheckSum = Transfer.YMODEMCheckSumType
  Transfer.YMODEMCheckSumType = "crc16"
End Sub
```

See Also File Transfer on page 2

Transfer.YMODEMPacketSize

VT, ANSI, SCO, and DG sessions only

Syntax **`Transfer.YMODEMPacketSize`**

Description Returns or sets the YMODEM-packet-size setting (integer). **`Transfer.YMODEMPacketSize`** accepts or returns either 128 or 1024.

Example

```
Sub Main
  Dim PktSize as Integer
  PktSize = Transfer.YMODEMPacketSize
  Transfer.YMODEMPacketSize = 1024
End Sub
```

See Also File Transfer on page 2

Transfer.YMODEMStreaming

VT, ANSI, SCO, and DG sessions only

Syntax `Transfer.YMODEMStreaming`

Description Returns or sets the YMODEM-streaming-mode setting (boolean).

Example

```
Sub Main
  Dim Streaming as Boolean
  Streaming = Transfer.YMODEMStreaming
  Transfer.YMODEMStreaming = True
End Sub
```

See Also File Transfer on page 2

Trim, Trim$, LTrim, LTrim$, RTrim, RTrim$

Syntax

```
Trim[$](string)
LTrim[$](string)
RTrim[$](string)
```

Description Returns a copy of the passed string expression (**string**) with leading and/or trailing spaces removed.

Trim returns a copy of the passed string expression (**string**) with both the leading and trailing spaces removed. **LTrim** returns **string** with the leading spaces removed, and **RTrim** returns **string** with the trailing spaces removed.

Trim$**, **LTrim$, and **RTrim$** return a **String**, whereas **Trim**, **LTrim**, and **RTrim** return a **String** variant.

Null is returned if **string** is **Null**.

Examples This first example uses the **Trim$** function to extract the nonblank part of a string and display it.

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  text$ = "      This is text           "
  tr$ = Trim$(text$)
  Session.Echo "Original =>" & text$ & "<=" & crlf & _
    "Trimmed =>" & tr$ & "<="
End Sub
```

This second example displays a right-justified string and its **LTrim** result.

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  a$ = "          <= This is a right-justified string"
  b$ = LTrim$(a$)
  Session.Echo a$ & crlf & b$
End Sub
```

This third example displays a left-justified string and its **RTrim** result.

```
Const crlf = Chr$(13) + Chr$(10)

Sub Main
  a$ = "This is a left-justified string.              "
  b$ = RTrim$(a$)
  Session.Echo a$ & "<=" & crlf & b$ & "<="
End Sub
```

Type

Syntax

```
Type username
  variable As type
  variable As type
  variable As type
  :
End Type
```

Description Creates a structure definition that can then be used with the **Dim** statement to declare variables of that type. The **username** field specifies the name of the structure that is used later with the **Dim** statement. Within a structure definition appear field descriptions in the format:

```
variable As type
```

where **variable** is the name of a field of the structure, and **type** is the data type for that variable. Any fundamental data type or previously declared user-defined data type can be used within the structure definition (structures within structures are allowed). Only fixed arrays can appear within structure definitions.

The **Type** statement can only appear outside of subroutine and function declarations.

When declaring strings within fixed-size types, it is useful to declare the strings as fixed-length. Fixed-length strings are stored within the structure itself rather than in the string space. For example, the following structure will always require 62 bytes of storage:

```
Type Person
  FirstName As String * 20
  LastName As String * 40
  Age As Integer
End Type
```

Note Fixed-length strings within structures are size-adjusted upward to an even byte boundary. Thus, a fixed-length string of length 5 will occupy 6 bytes of storage within the structure.

Example This example displays the use of the Type statement to create a structure representing the parts of a circle and assign values to them.

```
Type Circ
  mesg As String
  rad As Integer
```

```
  dia As Integer
  are As Double
  cir As Double
End Type'!
  Dim circle As Circ
  circle.rad = 5
  circle.dia = c
Sub Main
ircle.rad * 2
  circle.are = (circle.rad ^ 2) * Pi
  circle.cir = circle.dia * Pi
  circle.mesg = "The area of the circle is: " & circle.are
  Session.Echo circle.mesg
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

TypeName

Syntax `TypeName(varname)`

Description Returns the type name of the specified variable. The returned string can be any of the following:

Returned String	Returned If **`varname`** Is
`"String"`	A string.
`Objecttype`	A data object variable. In this case, **`objecttype`** is the name of the specific object type.
`"Integer"`	An integer.
`"Long"`	A long.
`"Single"`	A single.
`"Double"`	A double.
`"Currency"`	A currency value.
`"Date"`	A date value.
`"Boolean"`	A boolean value.
`"Error"`	An error value.
`"Empty"`	An uninitialized variable.
`"Null"`	A variant containing no valid data.
`"Object"`	A data or OLE automation object.
`"Unknown"`	An unknown type of OLE automation object.
`"Nothing"`	An uninitialized object variable.
`class`	A specific type of OLE automation object. In this case, **`class`** is the name of the object as known to OLE.

If **varname** is an array, then the returned string can be any of the above strings follows by a empty parenthesis. For example, **"Integer()"** would be returned for an array of integers.

If **varname** is an expression, then the expression is evaluated and a **String** representing the resultant data type is returned.

If **varname** is a collection, then **TypeName** returns the name of that object collection.

Example

```
Sub Foo(a As Variant)
  If VarType(a) <> ebInteger Then
    Session.Echo "Foo does not support " & TypeName(a) & " variables"
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

TypeOf

Syntax **TypeOf objectvariable Is objecttype**

Description Returns **True** if **objectvariable** is the specified type; **False** otherwise. This function is used within the **If...Then** statement to determine if a variable is of a particular type. This function is particularly useful for determining the type of OLE automation objects.

Example

```
Sub Main
  Dim a As Object
  Set a = CreateObject("Excel.Application")
  If TypeOf a Is "Application" Then
    Session.Echo "We have an Application object."
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

U

UBound

Syntax `UBound(ArrayVariable() [,dimension])`

Description Returns an **`Integer`** containing the upper bound of the specified dimension of the specified array variable. The **`dimension`** parameter is an integer that specifies the desired dimension. If not specified, then the upper bound of the first dimension is returned.

The **`UBound`** function can be used to find the upper bound of a dimension of an array returned by an OLE Automation method or property:

`UBound(object.property [,dimension])`

`UBound(object.method [,dimension])`

Examples

```
Sub Main
  Dim a(5 To 12)
  Dim b(2 To 100, 9 To 20)
  uba = UBound(a)
  ubb = UBound(b,2)
  Session.Echo "The upper bound of a is: " & uba & _
    " The upper bound of b is: " & ubb
'This example uses Lbound and Ubound to dimension a dynamic
'array to hold a copy of an array redimmed by the FileList
'statement.
Dim fl$()
FileList fl$,"*"
count = Ubound(fl$)
If ArrayDims(a) Then
  Redim nl$(Lbound(fl$) To Ubound(fl$))
  For x = 1 To count
    nl$(x) = fl$(x)
  Next x
  Session.Echo "The last element of the new array is: " & nl$(count)
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

UCase, UCase$

Syntax `UCase[$](string)`

Description Returns the uppercase equivalent of the specified string. **`UCase$`** returns a **`String`**, whereas **`UCase`** returns a **`String`** variant. **`Null`** is returned if **`string`** is **`Null`**.

Example

```
Sub Main
  a1$ = "this string was lowercase, but was converted."
  a2$ = UCase$(a1$)
  Session.Echo a2$
End Sub
```

See Also Character and String Manipulation on page 3

Unlock

See Lock, Unlock; Drive, Folder, and File Access on page 4.

User-Defined Types (topic)

`User-defined types` (UDTs) are structure definitions created using the **`Type`** statement. UDTs are equivalent to C language structures.

Declaring Structures

The **`Type`** statement is used to create a structure definition. Type declarations must appear outside the body of all subroutines and functions within a macro and are therefore global to an entire macro. Once defined, a UDT can be used to declare variables of that type using the **`Dim`**, **`Public`**, or **`Private`** statement. The following example defines a rectangle structure:

```
Type Rect
  left As Integer
  top As Integer
  right As Integer
  bottom As Integer
End Type
  :
Sub Main
  Dim r As Rect
    :
    r.left = 10
  End Sub
```

Any fundamental data type can be used as a structure member, including other user-defined types. Only fixed arrays can be used within structures.

Copying Structures

UDTs of the same type can be assigned to each other, copying the contents. No other standard operators can be applied to UDTs.

```
Dim r1 As Rect
Dim r2 As Rect
  :
r1 = r2
```

When copying structures of the same type, all strings in the source UDT are duplicated and references are placed into the target UDT.

The **LSet** statement can be used to copy a UDT variable of one type to another:

```
LSet variable1 = variable2
```

LSet cannot be used with UDTs containing variable-length strings. The smaller of the two structures determines how many bytes get copied.

Passing Structures

UDTs can be passed both to user-defined routines and to external routines, and they can be assigned. UDTs are always passed by reference. Since structures are always passed by reference, the **ByVal** keyword cannot be used when defining structure arguments passed to external routines (using **Declare**). The **ByVal** keyword can only be used with fundamental data types such as **Integer** and **String**.

Note Passing structures to external routines actually passes a far pointer to the data structure.

Size of Structures

The **Len** function can be used to determine the number of bytes occupied by a UDT:

```
Len(udt_variable_name)
```

Since strings are stored in the compiler's data space, only a reference (currently, 2 bytes) is stored within a structure. Thus, the **Len** function may seem to return incorrect information for structures containing strings.

V

Val

Syntax **`Val(string)`**

Description Converts a given string expression to a number. The **`string`** parameter can contain any of the following:

- Leading minus sign (for nonhexadecimal or octal numbers only)
- Hexadecimal number in the format **`&Hhexdigits`**
- Octal number in the format **`&Ooctaldigits`**
- Floating-point number, which can contain a decimal point and an optional exponent

Spaces, tabs, and line feeds are ignored.

If **`string`** does not contain a number, then 0 is returned.

The **`Val`** function continues to read characters from the string up to the first nonnumeric character.

The **`Val`** function always returns a double-precision floating-point value. This value is forced to the data type of the assigned variable.

Example

```
Sub Main
  a$ = InputBox$("Enter anything containing a number", _
    "Enter Number")
  b# = Val(a$)
  Session.Echo "The value is: " & b#
End Sub
```

See Also Character and String Manipulation on page 3

Variant (data type)

Syntax **`Variant`**

Description Used to declare variables that can hold one of many different types of data. During a variant's existence, the type of data contained within it can change. Variants can contain any of the following types of data:

Type of Data	Data Types
Numeric	Integer, long, single, double, boolean, date, currency.
Logical	Boolean.
Dates and times	Date.
String	String.
Object	Object.
No valid data	A variant with no valid data is considered null.
Uninitialized	An uninitialized variant is considered empty.

There is no type-declaration character for variants.

The number of significant digits representable by a variant depends on the type of data contained within the variant.

`Variant` is the default data type. If a variable is not explicitly declared with **`Dim`**, **`Public`**, or **`Private`**, and there is no type-declaration character (i.e., #, @, !, %, or &), then the variable is assumed to be **`Variant`**.

Determining the Subtype of a Variant

The following functions are used to query the type of data contained within a variant:

Function	Description
`VarType`	Returns a number representing the type of data contained within the variant.
`IsNumeric`	Returns **`True`** if a variant contains numeric data. The following are considered numeric: integer, long, single, double, date, boolean, currency. If a variant contains a string, this function returns **`True`** if the string can be converted to a number. If a variant contains an object whose default property is numeric, then **`IsNumeric`** returns **`True`**.
`IsObject`	Returns **`True`** if a variant contains an object.

Function	Description
`IsNull`	Returns **`True`** if a variant contains no valid data.
`IsEmpty`	Returns **`True`** if a variant is uninitialized.
`IsDate`	Returns True if a variant contains a date. If the variant contains a string, then this function returns True if the string can be converted to a date. If the variant contains an object, then this function returns True if the default property of that object can be converted to a date.

Assigning to Variants

Before a **`Variant`** has been assigned a value, it is considered empty. Thus, immediately after declaration, the **`VarType`** function will return **`ebEmpty`**. An uninitialized variant is 0 when used in numeric expressions and is a zero-length string when used within string expressions.

A **`Variant`** is **`Empty`** only after declaration and before assigning it a value. The only way for a **`Variant`** to become **`Empty`** after having received a value is for that variant to be assigned to another **`Variant`** containing **`Empty`**, for it to be assigned explicitly to the constant **`Empty`**, or for it to be erased using the **`Erase`** statement.

When a variant is assigned a value, it is also assigned that value's type. Thus, in all subsequent operations involving that variant, the variant will behave like the type of data it contains.

Operations on Variants

Normally, a **`Variant`** behaves just like the data it contains. One exception to this rule is that, in arithmetic operations, variants are automatically promoted when an overflow occurs. Consider the following statements:

```
Dim a As Integer,b As Integer,c As Integer
Dim x As Variant,y As Variant,z As Variant
a% = 32767
b% = 1
c% = a% + b%    'This will overflow.
x = 32767
y = 1
z = x + y    'z becomes a Long because of Integer overflow.
```

In the above example, the addition involving **`Integer`** variables overflows because the result (32768) overflows the legal range for integers. With **`Variant`** variables, on the other hand, the addition operator recognizes the overflow and automatically promotes the result to a **`Long`**.

Adding Variants

The **+** operator is defined as performing two functions: when passed strings, it concatenates them; when passed numbers, it adds the numbers.

With variants, the rules are complicated because the types of the variants are not known until execution time. If you use **+**, you may unintentionally perform the wrong operation.

It is recommended that you use the **&** operator if you intend to concatenate two **String** variants. This guarantees that string concatenation will be performed and not addition.

Variants That Contain No Data

A **Variant** can be set to a special value indicating that it contains no valid data by assigning the **Variant** to **Null**:

```
Dim a As Variant
a = Null
```

The only way that a **Variant** becomes **Null** is if you assign it as shown above.

The **Null** value can be useful for catching errors since its value propagates through an expression.

Variant Storage

Variants require 16 bytes of storage internally:

- A 2-byte type
- A 2-byte extended type for data objects
- 4 bytes of padding for alignment
- An 8-byte value

Unlike other data types, writing variants to **Binary** or **Random** files does not write 16 bytes. With variants, a 2-byte type is written, followed by the data (2 bytes for **Integer** and so on).

Disadvantages of Variants

The following list describes some disadvantages of variants:

- Using variants is slower than using the other fundamental data types (i.e., **Integer**, **Long**, **Single**, **Double**, **Date**, **Object**, **String**, **Currency**, and **Boolean**). Each operation involving a **Variant** requires examination of the variant's type.
- Variants require more storage than other data types (16 bytes as opposed to 8 bytes for a **Double**, 2 bytes for an **Integer**, and so on).
- Unpredictable behavior. You may write code to expect an **Integer** variant. At runtime, the variant may be automatically promoted to a **Long** variant, causing your code to break.

Passing Nonvariant Data to Routines Taking Variants

Passing nonvariant data to a routine that is declared to receive a variant by reference prevents that variant from changing type within that routine. For example:

```
Sub Foo(v As Variant)
  v = 50        'OK.
  v = "Hello, world."    'Get a type-mismatch error here!
End Sub

Sub Main
  Dim i As Integer
  Foo i        'Pass an integer by reference.
End Sub
```

In the above example, since an **Integer** is passed by reference (meaning that the caller can change the original value of the **Integer**), the caller must ensure that no attempt is made to change the variant's type.

Passing Variants to Routines Taking Nonvariants

Variant variables cannot be passed to routines that accept nonvariant data by reference, as demonstrated in the following example:

```
Sub Foo(i as Integer)
End Sub

Sub Main
  Dim a As Variant
  Foo a        'Compiler gives type-mismatch error here.
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

VarType

Syntax `VarType(varname)`

Description Returns an **Integer** representing the type of data in **varname**. The **varname** parameter is the name of any **Variant**. The following table shows the different values that can be returned by **VarType**:

Value	Constant	Data Type
0	**ebEmpty**	Uninitialized
1	**ebNull**	No valid data
2	**ebInteger**	Integer
3	**ebLong**	Long
4	**ebSingle**	Single
5	**ebDouble**	Double

Value	Constant	Data Type
6	**`ebCurrency`**	Currency
7	**`ebDate`**	Date
8	**`ebString`**	String
9	**`ebObject`**	OLE Automation object
10	**`ebError`**	User-defined error
11	**`ebBoolean`**	Boolean
12	**`ebVariant`**	Variant (not returned by this function)
13	**`ebDataObject`**	Non-OLE Object

When passed an object, the **`VarType`** function returns the type of the default property of that object. If the object has no default property, then either **`ebObject`** or **`ebDataObject`** is returned, depending on the type of variable.

Example

```
Sub Main
  Dim v As Variant
  v = 5&          'Set v to a Long.
  If VarType(v) = ebInteger Then
    Session.Echo "v is an Integer."
  ElseIf VarType(v) = ebLong Then
    Session.Echo "v is a Long."
  End If
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

W - X - Y

Weekday

Syntax `Weekday(date [,firstdayofweek])`

Description Returns an **`Integer`** value representing the day of the week given by date. Sunday is 1, Monday is 2, and so on.

Parameter	Description
`date`	Any expression representing a valid date.
`Firstdayofweek`	Indicates the first day of the week. If omitted, then Sunday is assumed (i.e., the constant **`ebSunday`** described below).

The **`firstdayofweek`** parameter, if specified, can be any of the following constants.

Constant	Value	Description
`ebUseSystem`	0	Use the system setting for **`firstdayofweek`**.
`ebSunday`	1	Sunday (the default)
`ebMonday`	2	Monday
`ebTuesday`	3	Tuesday
`ebWednesday`	4	Wednesday
`ebThursday`	5	Thursday
`ebFriday`	6	Friday
`ebSaturday`	7	Saturday

Example

```
Sub Main
  Dim a$(7)
  a$(1) = "Sunday"
  a$(2) = "Monday"
  a$(3) = "Tuesday"
  a$(4) = "Wednesday"
```

```
  a$(5) = "Thursday"
  a$(6) = "Friday"
  a$(7) = "Saturday"
Reprompt:
  bd = InputBox$("Please enter your birthday.","Enter Birthday")
  If Not(IsDate(bd)) Then Goto Reprompt
  dt = DateValue(bd)
  dw = WeekDay(dt)
  Session.Echo "You were born on day " & dw & ", which was a " & a$(dw)
End Sub
```

See Also Time and Date Access on page 17

While...Wend

Syntax

```
While condition
  [statements]
Wend
```

Description Repeats a statement or group of statements while a condition is **True**. The condition is initialized and then checked at the top of each iteration through the loop. Due to errors in program logic, you can inadvertently create infinite loops in your code. When you're running a macro within the macro editor, you can break out of an infinite loop by pressing Ctrl+Break.

Example

```
Sub Main
  x% = 0
  count% = 0
  While x% <> 1 And count% < 500
    x% = Rnd(1)
    If count% > 1000 Then
      Exit Sub
    Else
      count% = count% + 1
    End If
  Wend
  Session.Echo "The loop executed " & count% & " times."
End Sub
```

See Also Macro Control and Compilation on page 10

Width#

Syntax `Width# filenumber, width`

Description Specifies the line width for sequential files opened in either **Output** or **Append** mode. The **Width#** statement requires the following named parameters:

Parameter	Description
filenumber	Integer used to refer to the open file—the number passed to the **Open** statement.
Width	Integer between 0 to 255 inclusive specifying the new width. If **width** is 0, then no maximum line length is used.

When a file is initially opened, there is no limit to line length. This command forces all subsequent output to the specified file to use the specified value as the maximum line length.

The **`Width`** statement affects output in the following manner: if the column position is greater than 1 and the length of the text to be written to the file causes the column position to exceed the current line width, then the data is written on the next line.

The Width statement also affects output of the Print command when used with the Tab and Spc functions.

Example

```
Sub Main
  Width #1,80
End Sub
```

See Also Drive, Folder, and File Access on page 4

Word$

Syntax **`Word$(text$,first[,last])`**

Description Returns a **`String`** containing a single word or sequence of words between **`first`** and **`last`**. The **`Word$`** function requires the following parameters:

Parameter	Description
`text$`	String from which the sequence of words will be extracted.
`First`	Integer specifying the index of the first word in the sequence to return. If **`last`** is not specified, then only that word is returned.
`Last`	Integer specifying the index of the last word in the sequence to return. If **`last`** is specified, then all words between **`first`** and **`last`** will be returned, including all spaces, tabs, and end-of-lines that occur between those words.

Words are separated by any nonalphanumeric characters such as spaces, tabs, end-of-lines, and punctuation. Embedded null characters are treated as regular characters.

If **`first`** is greater than the number of words in **`text$`**, then a zero-length string is returned.

If **`last`** is greater than the number of words in **`text$`**, then all words from **`first`** to the end of the text are returned.

Example

```
Sub Main
  s$ = "My surname is Williams; Stuart is my given name."
  c$ = Word$(s$,5,6)
  Session.Echo "The extracted name is: " & c$
End Sub
```

See Also Character and String Manipulation on page 3

WordCount

Syntax `WordCount(text$)`

Description Returns an **`Integer`** representing the number of words in the specified text. Words are separated by spaces, tabs, and end-of-lines. Embedded null characters are treated as regular characters.

Example

```
Sub Main
  s$ = "My surname is Williams; Stuart is my given name."
  i% = WordCount(s$)
  Session.Echo "'" & s$ & "' has " & i% & " words."
End Sub
```

See Also Character and String Manipulation on page 3

Write#

Syntax `Write [#]filenumber [,expressionlist]`

Description Writes a list of expressions to a given sequential file. The file referenced by **`filenumber`** must be opened in either **`Output`** or **`Append`** mode. The **`filenumber`** parameter is an **`Integer`** used to refer to the open file—the number passed to the **`Open`** statement. The following summarizes how variables of different types are written:

Data Type	Description
Any numeric type	Written as text. There is no leading space, and the period is always used as the decimal separator.
String	Written as text, enclosed within quotes.
Empty	No data is written.
Null	Written as #NULL#.
Boolean	Written as **`#TRUE#`** or **`#FALSE#.`**
Date	Written using the universal date format: **`#YYYY-MM-DD HH:MM:SS#`**
User-defined errors	Written as **`#ERROR ErrorNumber#`**, where **`ErrorNumber`** is the value of the user-defined error. The word ERROR is not translated.

The **`Write`** statement outputs variables separated with commas. After writing each expression in the list, **`Write`** outputs an end-of-line.

The **`Write`** statement can only be used with files opened in **`Output`** or **`Append`** mode.

Example

```
Sub Main
  Open "test.dat" For Output Access Write As #1
  For x = 1 To 10
    r% = x * 10
     Write #1,x,r%
```

```
  Next x
  Close
  Open "test.dat" For Input Access Read As #1
  For x = 1 To 10
    Input #1,a%,b%
    mesg = mesg & "Record " & a% & ": " & b% & Basic.Eoln$
  Next x
  Session.Echo mesg
  Close
End Sub
```

See Also Drive, Folder, and File Access on page 4

WriteIni

Syntax **`WriteIni section$,ItemName$,value$[,filename$]`**

Description Writes a new value into an INI file. The **`WriteIni`** statement requires the following parameters:

Parameter	Description
`section$`	String specifying the section that contains the desired variables, such as "Windows." Section names are specified without the enclosing brackets.
`ItemName$`	String specifying which item from within the given section you want to change. If **`ItemName$`** is a zero-length string (""), then the entire section specified by **`section$`** is deleted.
`value$`	String specifying the new value for the given item. If **`value$`** is a zero-length string (""), then the item specified by **`ItemName$`** is deleted from the INI file.
`Filename$`	String specifying the name of the INI file.

If **`filename$`** is not specified, the win.ini file is used.

If the **`filename$`** parameter does not include a path, then this statement looks for INI files in the Windows directory.

Example

```
Sub Main
  WriteIni "Extensions","txt", _
    "c:\windows\notepad.exe ^.txt","win.ini"
End Sub
```

See Also Drive, Folder, and File Access on page 4

Xor

Syntax **`result = expression1 Xor expression2`**

Description Performs a logical or binary exclusion on two expressions. If both expressions are either **`Boolean`**, **`Boolean`** variants, or **`Null`** variants, then a logical exclusion is performed as follows:

If expression1 is	and expression2 is	then the result is
True	True	False
True	False	True
False	True	True
False	False	False

If either expression is **Null**, then **Null** is returned.

Binary Exclusion

If the two expressions are **Integer**, then a binary exclusion is performed, returning an **Integer** result. All other numeric types (including **Empty** variants) are converted to **Long**, and a binary exclusion is then performed, returning a **Long** result.

Binary exclusion forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions according to the following table:

If bit in expression1 is	and bit in expression2 is	the result is
1	1	0
0	1	1
1	0	1
0	0	0

Example

```
Sub Main
  For x = -1 To 0
    For y = -1 To 0
      z = x Xor y
      mesg = mesg & Format(x,"True/False") & " Xor "
      mesg = mesg & Format(y,"True/False") & " = "
      mesg = mesg & Format(z,"True/False") & Basic.Eoln$
    Next y
  Next x
  Session.Echo mesg
End Sub
```

See Also Keywords, Data Types, Operators, and Expressions on page 6

Year

Syntax `Year(date)`

Description Returns the year of the date encoded in the specified date parameter. The value returned is between 100 and 9999 inclusive. The **date** parameter is any expression representing a valid date.

Example

```
Sub Main
  tdate$ = Date$
  tyear! = Year(DateValue(tdate$))
  Session.Echo "The current year is: " & tyear$
End Sub
```

See Also Time and Date Access on page 17

PSL Equivalents for Methods and Properties

This table is provided for users of earlier SmarTerm versions, which relied on the Persoft Script Language (PSL). PSL has been replaced by the SmarTerm macro language, which is substantially similar to Visual Basic, but tailored for the SmarTerm user.

This table, like all the reference material in this book, is available in online help.

‰ Where the Macro Language side says "Not a one-for-one replacement," more than a single line of code is required to accomplish the translation.

PSL	Macro Language
`ABS`	`Abs`
`AND`	`And`
`ANSWER`	`Not a one-for-one replacement.`
`APPKEYBOARDMAP`	`Session.LoadKeyboardMap`
`ASC`	`Asc`
`ATEOF`	`Eof`
`AUXKEYBOARDMAP`	`Session.KeyboardMap`
`BUFFERFORMATTED`	`Session.BufferFormatted`
`BUFFERMODIFIED`	`Session.BufferModified`
`BUTTONPALETTE`	`Session.LoadSmarTermButtons`
`BUTTONPALETTE`	`Session.UnloadSmarTermButtons`
`CAPTURE`	`Session.Capture`
`CAPTURE SETUP`	`Session.CaptureFileHandling`
`CHAIN`	`Not a one-for-one replacement.`
`CHDIR`	`ChDir`
`CHDRIVE`	`ChDrive`

PSL	Macro Language
CHR$	Chr$
CIRCUIT CONNECT	Circuit.Connect
CIRCUIT DISCONNECT	Circuit.Disconnect
CIRCUIT SETUP	Circuit.Setup
CLOSE	Close
CLS	Session.ClearScreen
CMDLINE	Application.CommandLine
COLLECT	Session.Collect.Start
COLLECT	Session.Collect.Status
COLLECT	Session.Collect.CollectedCharacters
COLLECT	Session.Collect.Consume
COLLECT	Session.Collect.MaxCharacterCount
COLLECT	Session.Collect.TermString
COLLECT	Session.Collect.Reset
COLLECT	Session.Collect.TimeoutMS
COLLECT	Session.Collect.Timeout
COLLECT	Session.Collect.TermStringExact
COLLECT	Session.Collect
COLUMN	Session.Column
CONNECTED	Session.Connected
CURDIR$	CurDir$
CURMOUSEX	Session.MouseCol
CURMOUSEY	Session.MouseRow
DATE$	Date$
DDE _ ASSIGN	DDEPoke
DDE _ COMMAND	DDEExecute
DDE _ CONNECT, NEXTDDECHAN	DDEInitiate
DDE _ DISCONNECT	DDETerminate
DDE _ FETCH	DDERequest
DDESTATUS	Not a one-for-one replacement.
DIAL	Circuit.Connect (Modem Connection)
DIM	Dim
ECHO	Session.Echo
EMULATION$	Session.EmulationInfo
ENDCAPTURE	Session.EndCapture
ENVIRON$	Environ$
ERRORBOX	MsgBox
ESCREEN$	Session.NativeScreenText
EXECUTE	Shell
EXIT	Exit Sub

PSL	Macro Language
FIELD$	Session.FieldText
FIELDENDCOL	Session.FieldEndCol
FIELDENDROW	Session.FieldEndRow
FIELDMODIFIED	Session.FieldModified
FIELDSTARTCOL	Session.FieldStartCol
FIELDSTARTROW	Session.FieldStartRow
FILEEXISTS	FileExists
FILEOPEN	FileAttr
FILEPOS	Loc
FILESELECT$	SaveFilename
FILESELECT$	OpenFilename
FLISTBOX$	SelectBox
FUNCTION	Session.DoMenuFunction
GETPROFILE$	ReadIni$
GOSUB	GoSub
GOTO	Goto
HANGUP	Circuit.Disconnect (Modem Connection)
HEX$	Hex$
IF..THEN..ELSEIF..ELSE..ENDIF	If..Then..ElseIf..Else..End If
IN3270	Session.EmulationInfo(0)
INPUT	Input#
INPUT	Line Input#
INPUT$	InputBox
INPUT$	AskPassword$
INSERTMODE	Session.InsertMode
INSTR	InStr
INVOKE	Invoke
ISDDEOPEN	Not a one-for-one replacement.
ISFIELDMARK	Session.IsFieldMark
ISNUMERIC	Session.IsNumeric
ISPROTECTED	Session.IsProtected
KEYBOARDLOCKED	Session.KeyboardLocked
KEYWAIT	Session.Keywait.Reset
KEYWAIT	Session.Keywait.KeyType
KEYWAIT	Session.Keywait.Start
KEYWAIT	Session.Keywait.Value
KEYWAIT	Session.Keywait
KEYWAIT	Session.Keywait.KeyCount
KEYWAIT	Session.Keywait.MaxKeyCount
KEYWAIT	Session.Keywait.KeyCode

PSL	Macro Language
KEYWAIT	**Session.Keywait.Status**
KEYWAIT	**Session.Keywait.TimeOutMS**
KEYWAIT	**Session.Keywait.TimeOut**
LCASE$	**Lcase$**
LEFT$	**Left$**
LEN	**Len**
LET	**Let**
LISTBOX$	**SelectBox**
LTRIM$	**Ltrim$**
MAXIMIZE	**Session.WindowState = 2**
MCICMD	**Mci**
MESSAGEBOX	**MsgBox (statement)**
MID$	**Mid$**
MINIMIZE	**Session.WindowState = 0**
MOUSEX	**Session.InitialMouseCol**
MOUSEY	**Session.InitialMouseRow**
NEGATE	**Not**
NEXTDDECHAN	**Not a one-for-one replacement.**
NEXTFILENO	**FreeFile**
NOT	**Not**
OKBOX	**MsgBox**
OPEN	**Open**
OR	**Or**
PAGE	**Session.Page**
PAUSE	**Sleep**
PLAYWAVE	**Not a one-for-one replacement.**
POSITION	**Seek**
PRINT	**Print#**
PRODUCT$	**Application.Product**
PUTPROFILE	**WriteIni**
RESTORE	**Session.WindowState = 1**
RETURN	**Return**
RIGHT$	**Right$**
ROW	**Session.Row**
RTRIM$	**Rtrim$**
SCREEN$	**Session.ScreenText**
SELECTWAIT	**Session.StringWait.Status**
SELECTWAIT	**Session.StringWait.MaxCharacterCount**
SELECTWAIT	**Session.StringWait.TimeoutMS**
SELECTWAIT	**Session.StringWait.Timeout**

PSL	Macro Language
SELECTWAIT	Session.StringWait.MatchStringExact
SELECTWAIT	Session.StringWait.MatchString
SELECTWAIT	Session.StringWait.Start
SELECTWAIT	Session.StringWait.Reset
SELECTWAIT	Session.StringWait
SEND	Session.Send
SEND +keyword	Session.SendKey
SEND BINARY	Circuit.SendRawToHost
SEND LITERAL	Session.SendLiteral
SEND NORMAL	Session.Send
SET / RESET BLINK	Session.Blink
SET / RESET BOLD	Session.Bold
SET / RESET CONCEALED	Session.Concealed
SET / RESET CRITICAL	Session.Lockstep
SET / RESET FLASHICON	Application.FlashIcon
SET / RESET INTERPRET	Session.InterpretControls
SET / RESET INVERSE	Session.Inverse
SET / RESET KEYABORT	Not a one-for-one replacement.
SET / RESET LOCAL	Session.Online
SET / RESET NORMAL	Session.Normal
SET / RESET ONLINE	Session.Online
SET / RESET UNDERLINE	Session.Underline
SET / RESET WRAP	Session.Autowrap
SETFONTSIZE	Session.SetFontSize
SETTITLE	Session.Caption
SHARE	Public
SNAPALL	Session.ScreenToFile
STATUS	Not a one-for-one replacement.
STCONFIG	Session.ConfigInfo
STOP	End
STR$	Str$
STRING$	String$
SYSTEMTICKS	Timer * 1000
TERMINATE [SESSION]	Session.Close
TERMINATE ALL	Application.Quit
TIME$	Time$
TRANSFER COMMAND	Transfer.Command
TRANSFER PROTOCOL	Session.TransferProtocol
TRANSFER RECEIVEFILE	Transfer.ReceiveFile
TRANSFER SENDFILE	Transfer.SendFile

PSL	Macro Language
TRANSFER SETUP	**Transfer.Setup**
TRANSLATEBINARY	**Session.TranslateBinary**
TRANSLATETEXT	**Session.TranslateText**
TRANSMIT	**Session.TransmitFile**
TYPE	**Session.TypeFile**
UCASE$	**Ucase$**
USERHELP	**Application.UserHelpFile**
USERHELP	**Application.UserHelpMenu**
VAL	**Val**
VERSION	**Application.Version**
VERSION$	**Application.Version**
WAITFOR	**Session.EventWait.EventType**
WAITFOR	**Session.EventWait.Value**
WAITFOR	**Session.EventWait.EventCount**
WAITFOR	**Session.EventWait.Status**
WAITFOR	**Session.EventWait.Abort**
WAITFOR	**Session.EventWait.Start**
WAITFOR	**Session.EventWait.Reset**
WAITFOR	**Session.EventWait.TimeOut**
WAITFOR	**Session.EventWait.MaxEventCount**
WAITFOR	**Session.EventWait**
WAITFOR	**Session.EventWait.TimeoutMS**
WARNINGLEVEL	**Circuit.SuppressConnectErrorDialog**
WHILE/WEND	**While .. Wend**
WINSTATE	**Session.WindowState**
XOR	**Xor**

Error Messages

This section contains listings of all the runtime errors. It is divided into two subsections, the first describing error messages compatible with "standard" Basic as implemented by Microsoft Visual Basic and the second describing error messages specific to the macro compiler.

A few error messages contain placeholders which are replaced to form the completed runtime error message. These placeholders appear in the following list as the italicized word *placeholder*.

Visual Basic Compatible error messages

Error Number	Error Message
3	Return without GoSub
5	Invalid procedure call
6	Overflow
7	Out of memory
9	Subscript out of range
10	This array is fixed or temporarily locked
11	Division by zero
13	Type mismatch
14	Out of string space
18	User interrupt occurred
19	No Resume
20	Resume without error
26	Dialog needs End Dialog or push button
28	Out of stack space
35	Sub or Function not defined
48	Error in loading DLL

Error Number	Error Message
49	Bad DLL calling convention
51	Internal error
52	Bad file name or number
53	File not found
54	Bad file mode
55	File already open
57	Device I/O error
58	File already exists
59	Bad record length
61	Disk full
62	Input past end of file
63	Bad record number
64	Bad file name
67	Too many files
68	Device unavailable
70	Permission denied
71	Disk not ready
74	Can't rename with different drive
75	Path/File access error
76	Path not found
91	Object variable or With block variable not set
93	Invalid pattern string
94	Invalid use of Null
139	Only one user dialog may be up at any time
140	Dialog control identifier does not match any current control
141	The **`placeholder`** statement is not available on this dialog control type
143	The dialog control with the focus may not be hidden or disabled
144	Focus may not be set to a hidden or disabled control
150	Dialog control identifier is already defined
163	This statement can only be used when a user dialog is active
260	No timer available
281	No more DDE channels
282	No foreign application responded to a DDE initiate
283	Multiple applications responded to a DDE initiate

Error Number	Error Message
285	Foreign application won't perform DDE method or operation
286	Timeout while waiting for DDE response
287	User pressed Escape key during DDE operation
288	Destination is busy
289	Data not provided in DDE operation
290	Data in wrong format
291	Foreign application quit
292	DDE conversation closed or changed
295	Message queue filled; DDE message lost
298	DDE requires ddeml.dll
380	Invalid property value
423	Property or method not found
424	Object required
429	OLE Automation server can't create object
430	Class doesn't support OLE Automation
431	OLE Automation server cannot load file
432	File name or class name not found during OLE Automation operation
438	Object doesn't support this property or method
440	OLE Automation error
442	Connection to type library or object library for remote process has been lost. Press OK for dialog to remove reference.
443	Object does not have a default value
445	Object doesn't support this action
446	Object doesn't support named arguments
447	Object doesn't support current locale setting
448	Named argument not found
449	Argument not optional
450	Wrong number of arguments or invalid property assignment
451	Object not a collection
452	Invalid ordinal
453	Specified DLL function not found
454	Code resource not found
455	Code resource lock error
460	Invalid Clipboard format
481	Invalid picture

Error Number	Error Message
520	Can't empty clipboard
521	Can't open clipboard
600	Set value not allowed on collections
601	Get value not allowed on collections
603	ODBC - SQLAllocEnv failure
604	ODBC - SQLAllocConnect failure
608	ODBC - SQLFreeConnect error
610	ODBC - SQLAllocStmt failure
3129	Invalid SQL statement; expected 'DELETE', 'INSERT', 'PROCEDURE', 'SELECT', or 'UPDATE'
3146	ODBC - call failed
3148	ODBC - connection failed
3276	Invalid database ID

Compiler-Specific error messages

Number	Error Message
800	Incorrect Windows version
801	Too many dimensions
802	Can't find window
803	Can't find menu item
804	Another queue is being flushed
805	Can't find control
806	Bad channel number
807	Requested data not available
808	Can't create pop-up menu
810	Command failed
811	Network error
812	Network function not supported
813	Bad password
814	Network access denied
815	Network function busy
816	Queue overflow
817	Too many dialog controls
818	Can't find list box/combo box item

Number	Error Message
819	Control is disabled
820	Window is disabled
821	Can't write to INI file
822	Can't read from INI file
823	Can't copy file onto itself
824	OLE Automation unknown object name
825	Redimension of a fixed array
826	Can't load and initialize extension
827	Can't find extension
828	Unsupported function or statement
829	Can't find ODBC libraries
830	OLE Automation Lbound or Ubound on non-Array value
831	Incorrect definition for dialog procedure
832	Incorrect number of arguments for intermodule call
833	OLE Automation object does not exist
834	Access to OLE Automation object denied
835	OLE initialization error
836	OLE Automation method returned unsupported type
837	OLE Automation method did not return a value

Compiler errors

The following table contains a list of all the errors generated by the macro compiler. With some errors, the compiler changes placeholders within the error to text from the macro being compiled. These placeholders are represented in this table by the word `placeholder`.

Number	Error Message
1	Variable Required - Can't assign to this expression
2	Letter range must be in ascending order
3	Redefinition of default type
4	Out of storage for variables
5	Type-character doesn't match defined type
6	Expression too complex
7	Cannot assign whole array
8	Assignment variable and expression are different types

Number	Error Message
9	No type-characters allowed on a function with an explicit type
10	Array type mismatch in parameter
11	Array type expected for parameter
12	Array type unexpected for parameter
13	Integer expression expected for an array index
14	Integer expression expected
15	String expression expected
16	Identifier is already a user defined type
17	Property value is the incorrect type
18	Left of "." must be an object, structure, or dialog
19	Invalid string operator
20	Can't apply operator to array type
21	Operator type mismatch
22	"**`placeholder`**" is not a variable
23	"**`placeholder`**" is not a array variable or a function
24	Unknown **`placeholder`** "**`placeholder`**"
25	Out of memory
26	**`placeholder`**: Too many parameters encountered
27	**`placeholder`**: Missing parameter(s)
28	**`placeholder`**: Type mismatch in parameter **`placeholder`**
29	Missing label "**`placeholder`**"
30	Too many nested statements
31	Encountered new-line in string
32	Overflow in decimal value
33	Overflow in hex value
34	Overflow in octal value
35	Expression is not constant
36	Not inside a **`Do`** statement
37	No type-characters allowed on parameters with explicit type
39	Can't pass an array by value
40	"**`placeholder`**" is already declared as a parameter
41	Variable name used as label name
42	Duplicate label
43	Not inside a function

Number	Error Message
44	Not inside a sub
46	Can't assign to function
47	Identifier is already a variable
48	Unknown type
49	Variable is not an array type
50	Can't redimension an array to a different type
51	Identifier is not a string array variable
52	0 expected
54	`placeholder` is not an assignable property of the object
55	Integer expression expected for file number
56	`placeholder` is not a method of the object
57	`placeholder` is not a property of the object
58	Expecting 0 or 1
59	Boolean expression expected
60	Numeric expression expected
61	Numeric type `FOR` variable expected
62	`For...Next` variable mismatch
63	Out of string storage space
64	Out of identifier storage space
68	Division by zero
69	Overflow in expression
70	Floating-point expression expected
72	Invalid floating-point operator
74	Single character expected
75	Subroutine identifier can't have a type-declaration character
76	Macro is too large to be compiled
77	Variable type expected
78	Can't evaluate expression
79	Can't assign to user or dialog type variable
80	Maximum string length exceeded
81	Identifier name already in use as another type
84	Operator cannot be used on an object
85	`placeholder` is not a property or method of the object
86	Type-character not allowed on label

Number	Error Message
87	Type-character mismatch on routine **`placeholder`**
88	Destination name is already a constant
89	Can't assign to constant
91	Identifier too long
92	Expecting string or structure expression
93	Can't assign to expression
94	Dialog and Object types are not supported in this context
95	Array expression not supported as parameter
96	Dialogs, objects, and structures expressions are not supported as a parameter
97	Invalid numeric operator
98	Invalid structure element name following "."
99	Access value can't be used with specified mode
101	Invalid operator for object
102	Can't **`LSet`** a type with a variable-length string
103	Syntax error
104	**`placeholder`** is not a method of the object
105	No members defined
106	Duplicate type member
107	Set is for object assignments
109	Invalid character in octal number
110	Invalid numeric prefix: expecting **`&H`** or **`&O`**
111	End-of-macro encountered in comment: expecting **`*/`**
112	Misplaced line continuation
113	Invalid escape sequence
114	Missing End Inline
115	Statement expected
116	ByRef argument mismatch
117	Integer overflow
118	Long overflow
119	Single overflow
120	Double overflow
121	Currency overflow
122	Optional argument must be Variant
123	Parameter must be optional

Number	Error Message
124	Parameter is not optional
125	Expected: `Lib`
126	Illegal external function return type
127	Illegal function return type
128	Variable not defined
129	No default property for the object
130	The object does not have an assignable default property
131	Parameters cannot be fixed length strings
132	Invalid length for a fixed length string
133	Return type is different from a prior declaration
134	Private variable too large. Storage space exceeded
135	Public variables too large. Storage space exceeded
136	No type-characters allowed on variable defined with explicit type
137	Missing parameters are not allowed when using named parameters
138	An unnamed parameter was found following a named parameter
139	Unknown parameter name: `placeholder`
140	Duplicate parameter name: `placeholder`
141	Expecting: `#If`, `#ElseIf`, `#Else`, `#End If`, or `#Const`
142	Invalid preprocessor directive
143	Expecting preprocessor variable
144	Expecting: `=`
145	Expecting: `[end of line]`
146	Expecting: **<expression>**
148	Expecting: `)`
149	Unexpected value
150	Expecting: `#End If`
151	Expecting: `Then`
152	Missing `#End If`
153	`#Else` encountered without `#If`
154	`#ElseIf` encountered without `#If`
155	`#End If` encountered without `#If`
156	Invalid use of `Null`
157	Type mismatch
158	Not a number

Number	Error Message
159	Duplicate subroutine function
160	Duplicate function definition
161	MBCS characters not allowed in identifiers
162	Out of range
163	Invalid date
164	Date overflow
165	Expecting: **`<identifier>`**
166	Constant type and expression are different types
167	Invalid use of **`New`**

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

P

Q

R

S

T

U

V

W

X

Y

Z