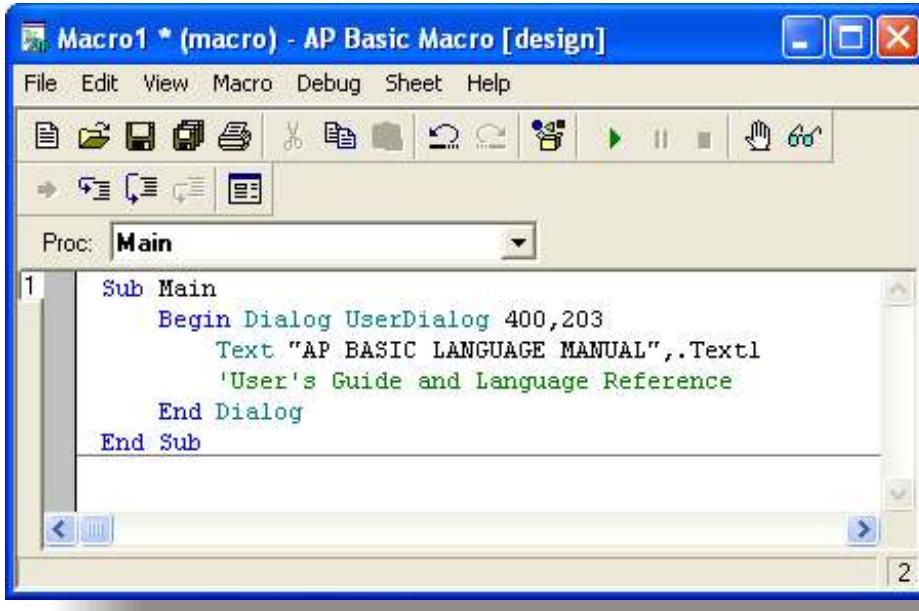
AP Basic

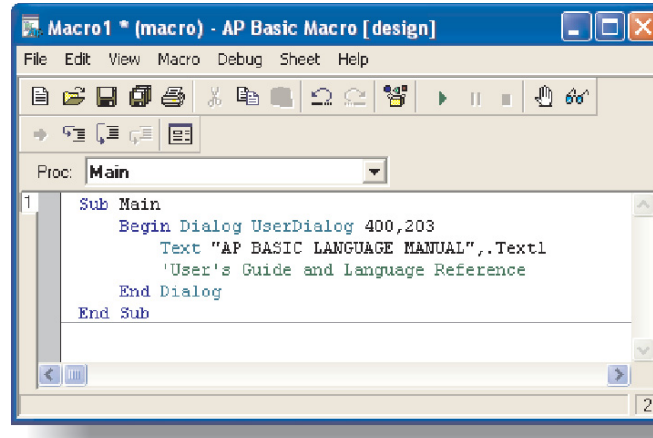
AP Basic Language Manual

System Automation



Audio precision®

AP Basic Language Manual



User's Guide and Language Reference for AP Basic, the programming language for Audio Precision test and measurement instruments

Copyright © 2003 Audio Precision, Inc.

All rights reserved.

PN 8211.0089 Revision 4

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the publisher.

Audio Precision®, System One®, System Two™, System Two Cascade™, System Two Cascade *Plus*™, Cascade™, Cascade *Plus*™, System One + DSP™, System Two + DSP™, Dual Domain®, FASTTEST®, APWIN™, ATS™ and ATS-2™are trademarks of Audio Precision, Inc. Windows is a trademark of Microsoft Corporation.

5750 SW Arctic Drive
Beaverton, Oregon 97005
Tel: 503-627-0832
Fax: 503-641-8906
US Toll Free: 1-800-231-7350
email: info@audioprecision.com
Web: audioprecision.com

Printed in the United States of America

III0820161119

Contents

Chapter 1
Introduction

Welcome to the *AP Basic Language Manual*, your guide to creating custom test programs for Audio Precision's PC-controlled measurement instruments, including

- System One
- System Two
- System Two Cascade
- System Two Cascade *Plus*
- the 2700 series, and
- the ATS-2.

These will be referred to collectively as "instruments" or "systems" throughout this guide.

AP Basic is a powerful and easy-to-use programming language compatible with Microsoft's Visual Basic for Applications (VBA). In this book, you'll learn how to create AP Basic programs called *macros* (sometimes called *procedures*) that can load and run tests, automate repetitive tasks, and add custom features and functions to your system's control software to suit your measurement needs.

AP Basic macros are lists of commands that tell the control software (APWIN, ATS, AP2700, etc.) what to do. Included with AP Basic are many *extension commands* you can use in your programs to automate control of your Audio Precision instrument. You do not need to develop any special commands to automate the control software or the instrument; all of these commands are available when you begin using AP Basic.

One of the most exciting features in AP Basic is its support of *ActiveX Automation.* ActiveX Automation is used with Microsoft Windows to allow ActiveX-compliant applications to share information. Using the ActiveX Automation features in AP Basic it is possible, for example, to take the results from a system measurement, move the data into any Microsoft Excel spreadsheet where it can be further manipulated, then take these results into Microsoft Word where they can be inserted into a report form. All of this can

be automated and run entirely from within AP Basic. The results of your Word document can even be printed from inside AP Basic.

NOTE: Earlier software versions and older documentation will refer to an Microsoft Windows protocol called OLE Automation, *which shares many features with ActiveX Automation and accomplished the same tasks in AP Basic.*

All of this power and functionality might lead you to think AP Basic is a difficult and complex programming language. In fact, AP Basic is one of the easiest development environments to use. Even if you have never programmed before, you will be surprised how quickly you will begin developing interesting and powerful programs.

AP Basic Documentation

AP Basic Language Manual

This book provides an introduction to programming in AP Basic. Chapters 1–5 are intended as a tutorial to help beginning users understand what AP Basic is and how to use it to develop programs.

Chapter 6 is a Language Reference and lists the generic commands available in AP Basic. These are the same commands you will find available in any Visual Basic (VB) compatible application.

AP Basic Extensions

Extensions to the generic commands detailed in Chapter 6 are documented in Basic Extensions Reference manuals specific to each Audio Precision instrument. Extension Reference manuals include:

- *AP Basic Extensions Reference for System One*
- *AP Basic Extensions Reference for System Two*
- *AP Basic Extensions Reference for System Two Cascade*
- *AP Basic Extensions Reference for System Two Cascade Plus*
- *AP Basic Extensions Reference for 2700 Series Instruments*
- *AP Basic Extensions Reference for ATS-2*

Chapter Overviews

Chapter 1 provides a general overview of Audio Precision control software to help the first-time user get started quickly.

Chapter 2 provides an introduction to the fundamentals of AP Basic. Several of the key concepts in Visual Basic are introduced, including *objects*, *methods* and *properties*, and the use of macros.

Chapter 3 moves beyond the concepts of Visual Basic and jumps into the fundamentals of writing a program. Working from a simple example, each of the key elements of a program is introduced and discussed. Some of the key topics discussed in this chapter include the structure of a program, syntax, and an introduction to commonly used commands.

Chapter 4 describes how to test and debug a program. AP Basic provides a number of tools to assist in verifying correct operation of a program. Additional topics include tips for simplifying the debugging process, common programming mistakes to avoid, and error handling.

Chapter 5 provides an introduction to the AP Basic Dialog Editor. The Dialog Editor provides an easy way of creating a user interface consisting of menus, and other dialogs that an operator can interact with to control your program.

Chapter 6 is a listing of generic commands available whenever you use AP Basic, regardless of instrument. These commands are used by all applications which utilize Visual Basic-compatible commands.

Manual Conventions

This manual uses the following typographic conventions.

Example	Description
event, var, arg	For the syntax part of each command, italicized words indicate placeholders where the user must enter additional information.
FILENAME.TXT	Words in all CAPITAL letters indicate file names.

`Sub Main`
`  AP.Gen.Amp = 1.0`
`End Sub`
	This font is used in all example macros and code modules.

[expressionlist]	In syntax, items inside square brackets are optional.
{While | Until}	In syntax, braces and a vertical bar indicate a choice between two or more items.

Command	For the syntax part of each command, the bold characters identify the part of the command that must be entered.

`AP.Prompt. _`
`  Text "This _`
`  is just an _`
`  example."`
	The line continue character (_) is used to indicate that the code from one line to the next should be typed on one line.

Terminology

Audio Precision has used the term *procedure* since our first product to identify a facility that will automatically run a sequence of tests; in other words, an automation *program* or *script*. This was in line with test and measurement industry usage, where the process of performing one or more tests or measurements has long been called a *test procedure*.

However, software programmers refer to such programs and scripts as *macros*, reserving the term *procedure* to identify specific parts of programs or subprograms.

Audio Precision is now adopting the term macro as the name for the AP Basic programs that automate our control software, and we are deprecating the use of the term procedure for this use.

In short, what were called *procedures* in older Audio Precision documentation and user interface are now called *macros*. However, you will find embedded references, folder names, etc. that continue to use the term *procedure*.

Sample Files and Examples

We have designed and written many sample macros for your instrument that you may choose to study or include as part of macros you may write. These samples are available in the Samples area under Support at our Web site at audioprecision.com.

AP Basic macros for System One, System Two, System Two Cascade and 2700 series instruments have the filename extension *.apb; AP Basic macros for the ATS-2 have the filename extension *.atsb.

These samples are excellent learning tools and are representative of the type of programs you are likely to develop. You can load these macros into the Macro Editor where you can edit them or even use them in part or entirely within your own program.

Using Online Help

Audio Precision includes extensive online help with the control software to assist you in developing AP Basic programs. Help is accessible in the following ways:

- Choose **Help** from the Main menu in the control software. If you have already opened the Macro Editor panel, you can select between AP Basic Language, AP Basic Extensions, or Macro Editor help.
- Highlight a command or keyword in the Macro Editor and press **F1** for context-sensitive help.
- Click the **Browse Object** button on the Macro Editor toolbar, and then select the method or property you need information about. The Object Browser provides information about all of the classes and objects available in the control software.

- Highlight a specific AP Basic extension command and click the **Browse Object** button on the Macro Editor toolbar for information about the methods and properties of the command.

Getting Started In AP Basic

AP Basic is automatically installed on your computer when you install the Audio Precision control software for your instrument. There are no extra installation steps necessary to use AP Basic. See the "Getting Started" manual included with your instrument for instruction on installing your control software.

To begin using AP Basic, open the Macro Editor panel in the control software. Open this panel by choosing **Macro Editor** in the Panels menu or by clicking the Macro Editor button on the toolbar. See Figure 1.

Macro Editor Overview

The Macro Editor offers a number of menu options and buttons to make it easier to use.

The Main menu runs across the top of the Macro Editor, offering access to all the editor functions through seven submenus. If you right click within the edit window, a popup menu will appear, offering the same choices as the Main menu.

Below the Main menu is the Toolbar, with one-button access to many of the editor functions. Each of the buttons is explained in detail in the on-line help. You can also get information about a specific button by pausing the mouse pointer over a button for a moment. A Tool Tip will pop up indicating the purpose of the button.

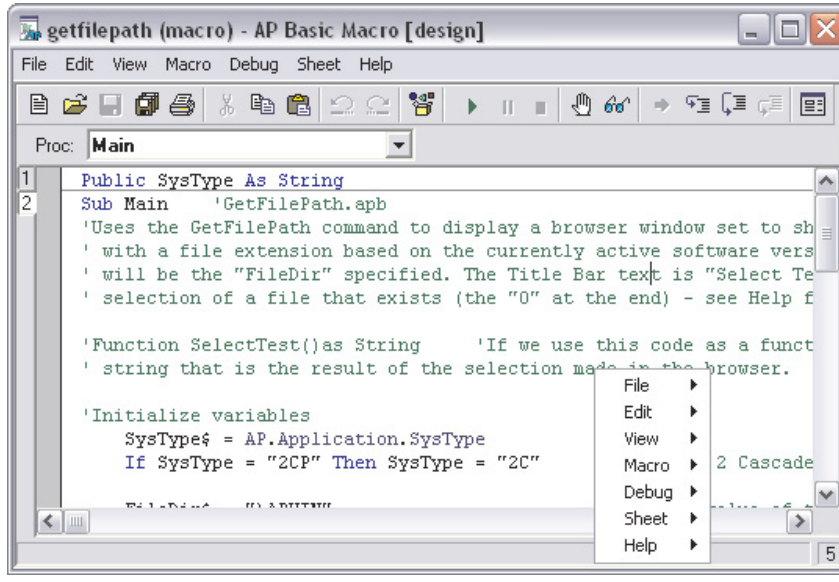


Figure 1. The Macro Editor

Click the mouse on any of the available menu options to select the option you want. To open an AP Basic macro, click the **Open** button or choose **File > Open**.

Sheets

You can use the Macro Editor to open several macros at one time. Each time a macro is opened, a new *sheet* is created and the macro is placed on the sheet. You can select between sheets by clicking the number on the sheet toolbar (running along the left edge of the Macro Editor, as shown in Figure 1) corresponding to the macro you want.. This enables you to quickly switch between macros when you want to cut and paste code. Close a sheet by double-clicking on the sheet number or by choosing **Sheet > Close**.

Once you have loaded or entered a macro, you can run it by clicking the **Start/Resume** button on the toolbar.

When you run a macro, AP Basic will execute the commands that make up the macro. If you have several sheets open at one time, AP Basic will only run the macro that is currently shown when the **Start/Resume** button is clicked. To run a different open macro, select the sheet number of the macro by clicking on the sheet toolbar on the left side of the Macro Editor, as shown in Figure 1.

The Macro Editor can also be used to diagnose and fix errors in your program. To use the debug features choose **Debug** from the Main menu or click

one of the debug buttons on the toolbar. See Chapter 4 for detailed information on testing and debugging macros.

Figure 2. The Debug Buttons: Show Current Statement, Step In, Step Over, Step Out.

Entering and Editing Code with the Macro Editor

In programming, the raw text that comprise the program is called *code*. To enter new code (text) or edit existing code with the Macro Editor, use the mouse to position the cursor to where you want to begin, and start typing. You will find the Macro Editor operates much like other text editors. You can cut, copy and paste code using the **Ctrl-X**, **Ctrl-C** and **Ctrl-V** hot keys that are standard in Microsoft Windows, or you can choose **Edit > Cut** and **Edit > Paste** from the Main menu.

It is also possible to copy text from a different Windows application and paste it onto a sheet. For example, you can copy sample code fragments from the AP Basic Help screen and paste these into your program.

To learn more about editing code or how to use the Macro Editor in general, choose **Help > Editor Help**. See Chapter 3 for detailed information on writing a macro.

Find Out More About Visual Basic

There are several good references available to help you learn Visual Basic. We recommend you consider the following:

- *Running Visual Basic for Windows* by Ross Nelson
- *The Microsoft Visual Basic Programmer's Guide*
- *The Microsoft Excel Visual Basic User's Guide*
- *The Microsoft Word Developer's Kit 6.0*
  This manual provides command reference and other information needed to communicate with Microsoft Word in order to produce reports or other documentation via OLE automation.

... for Experienced VB Programmers

An experienced Visual Basic programmer may need to understand in what ways AP Basic differs from Microsoft Visual Basic.

Currently, Visual Basic exists in three editions: a Professional edition, a Standard edition, and an Applications edition, known as Visual Basic for Applications or VBA. VBA is a subset of the Professional and Standard editions of VB and is designed to be embedded within an application.

AP Basic is compatible only with VBA. It can only be run from within the control software and does not include the forms package included with the full Professional or Standard editions of Visual Basic.

For information on specific AP Basic commands that may differ from standard Microsoft Visual Basic, consult the online help.

Chapter 2

Fundamentals of AP Basic

This chapter begins our discussion of AP Basic. First we look at subroutines, one of the most fundamental elements in an AP Basic program. Subroutines are used to group commands together that when combined perform a specific task. Collections of subroutines are often organized to form a complete macro. We will look at how subroutines are structured and how they are used in programs.

In the second half of this chapter, we move from subroutines to study how data is represented in AP Basic. *Objects* are introduced as a way to organize collections of code and data that are related. *Properties* are characteristics of objects that can be used to change the attributes of an object. *Methods* are another characteristic of objects that can perform a function. Although subroutines and objects may at first seem to be related in how they group together common commands and data, they are distinctly different parts of a program. In this chapter we will examine subroutines and objects more closely.

A complete discussion of the different parts of a program is postponed until Chapter 3. If you discover while reading this chapter that you need an example of a program to work from, you can flip to the beginning of Chapter 3 where a complete program example is given.

What is an AP Basic Program?

A program is a collection of one or more AP Basic macros. Each macro can contain zero or more subroutines. Each subroutine contains commands that do something useful.

For example, a program might be written to load and run a number of tests in the control software. Another program might be written to combine the results of several tests and extract common trends in the data. Yet another program might offer a dialog box from which a user can select between different programs to run. There is no requirement on what a program must do other than it must consist of valid AP Basic commands that can be executed.

A program can be as big or as small as you choose. Since programmers often want to combine several different operations into one program, programs

tend to become large and complex fairly quickly. Subroutines are used to help organize programs into sections of similar code.

Using Subroutines

Subroutines are collections of AP Basic commands that are executed as a unit. When the control software executes a subroutine, it starts with the first command in the subroutine and proceeds from top to bottom, one line at a time. A well written subroutine should accomplish a single task. For example, a subroutine might load and run a test, alter how the control software is configured, or collect information from a user. Complicated tasks should be broken down into several subroutines. A complete program may use any number of subroutines.

There are three main benefits of programming with subroutines.

- Subroutines allow you to break your application into separate, logical elements, each of which you can understand and debug more easily.
- Subroutines can simplify and condense code by combining repeated or common tasks into just one piece of code.
- Subroutines used in one program can be copied and used as building blocks for another program. Once you have a subroutine that works well, you will want to use this subroutine in other programs rather than spending the time to re-write code.

AP Basic uses two main types of subroutines: subroutines and functions subroutines. A subroutine performs a specific task but does not return a result. A function is similar to a subroutine except that it can return a result. Each of these types of subroutines is discussed in more detail below.

Elements of a Subroutine

Before exploring the differences between subroutines and functions, it's instructive to look at the elements common to all subroutines. A clear understanding of a subroutine's structure will help you avoid common mistakes that often frustrate beginning programmers. It will also help you to read and understand other examples of AP Basic code.

All subroutines have the following parts:

- Begin and End statements at the top and bottom of the subroutine, respectively.
- A label that uniquely identifies the subroutine.
- Arguments that follow the subroutine label.
- AP Basic code.

The beginning and end statements for a sub subroutine follow the general form:

```
Sub Label(arguments)
 …
End Sub
```

The first line of a subroutine always begins with the Sub statement, the label of the subroutine, and a set of parentheses in which arguments are placed. If the subroutine doesn't require any arguments, the parenthesis are not required. The label of a subroutine is a unique name you choose that allows you to refer to the subroutine. Typically, you should choose subroutine labels that describe what the subroutine does. For example, a subroutine that prompts the user for their initials might use the following first line:

```
Sub PromptForInitials ()
```

A subroutine label can be almost any combination of characters and numbers except that it must start with a character and not contain any spaces.

Arguments

The arguments that follow a subroutine label allow the programmer to pass specific information to the subroutine. During a typical program, a subroutine may be executed from several different points in the code, but the data used by the subroutine may need to change. Arguments provide a means to vary the information used in a subroutine. The topic of arguments and how and when to use them in subroutines is not difficult but has some subtleties and variations that are beyond the scope of this tutorial. Refer to any of the Visual Basic programming manuals mentioned in the introductory chapter for more information on using arguments in subroutines.

The bulk of a subroutine consists of the code. These are commands that tell AP Basic what to do. There are a large number of commands available in AP Basic and almost all of them may be used in subroutines. Any command you want to use in a subroutine must be placed within the Sub and End Sub statements.

Technically, the number of commands you can place in a subroutine is quite large; practically, however, you will want to limit the number of commands in any one subroutine. Your goal when writing a subroutine should be to use only the commands you need to accomplish a specific task. If your program needs to do several different tasks, then you should write several different subrou-

tines, one for each task. It is much easier to understand and debug small blocks of code than to try and sift your way through an unnecessarily large and complex subroutine.

The second type of subroutines used in AP Basic are functions. They are similar to subroutines and follow the general form:

```
Function FunctionLabel(arguments)
  …
End Function
```

Functions are written in the same way as subroutines but with one important difference. The commands inside a function should assign a return value to the label you gave the function. When the function is finished executing, AP Basic will return the value assigned to the function label to the line of code that called the function.

For example, you could write a function that calculates the value of a number in decibels (dB).

```
Function TodB (num)
 TodB = 20*Log10(num)
End Function
```

You call a function the same way you call any of the built-in functions in AP Basic.

```
result = TodB (data)
```

Here is the previous example together with sample code that calls the function. In this example, two channels of data are converted, one element at a time, to a dB format.

```
Sub convertData(numPoints)
 For n = 0 To numPoints
 dataCh1(n) = TodB(dataCh1(n))
 dataCh2(n) = TodB(dataCh2(n))
 Next n
End Sub
```

```
Function TodB (num)
 TodB = 20*Log10(num)
End Function
```

The techniques for calling all types of subroutines are discussed in the section **Calling Subroutines** beginning on page 14.

Subroutines and functions are the building blocks of any AP Basic application. They can be combined and used in any way you choose to make your application useful. The next section looks more closely at some of the different ways to use subroutines.

How to Use Subroutines

In order to develop an AP Basic program, you must first understand how to use subroutines. In this section we look at some of the different uses of subroutines and how they can be combined to form a menu.

One key use of subroutines is to define where program execution begins. A typical AP Basic program may have several different subroutine and functions. In order to begin running the program, AP Basic must know which of these to start from.

In AP Basic, program execution starts with the first line of code in the Main subroutine. The Main subroutine is just like any other subroutine. You can use any commands you want in any order you choose. What's special about the Main subroutine is that execution will always start with the first line of code. Here is an example of a Main subroutine.

```
Sub Main
   Call runTest()
   Call processResults()
   Call printResults()
End Sub
```

In this example, the only code in the Main sub subroutine are calls to other subroutines. In this way, the Main subroutine is used to organize how program execution flows through the code.

All AP Basic programs you write will need to have a Main subroutine. If you try to run your program without a Main subroutine, or with two subroutines using the Main label, you will get an error.

Unless your program is very simple, you're likely to want to use several subroutines in addition to the Main subroutine. As shown below, you access additional subroutines and functions by calling them from within another subroutine.

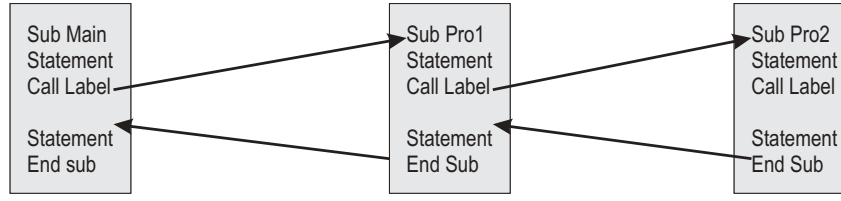


Figure 3.

Calling Subroutines

The techniques for calling subroutines vary, depending on the type of subroutine, where it's located, and how it's used.

A subroutine is called by a stand-alone statement. Unlike a function, a subroutine does not return a value, but can modify the values of any variables passed to it.

There are two ways to call subroutines.

```
Call MySubroutine (argument1, argument2)
```

-OR-

```
Mysubroutine argument1, argument2
```

Note that when the Call syntax is used, the arguments passed to the subroutine must be enclosed in parentheses. When the Call syntax is not used, the parentheses can be omitted.

A call to a function is made in the same way you call any intrinsic Visual Basic function, like Log10, that is, by using its name in an expression.

```
'The following statement calls the TodB function
result = TodB (data)
```

It is also possible to call a function just like you would a subroutine.

```
Call TodB (data)
```

-OR-

```
TodB data
```

When functions are called this way, AP Basic discards the return value.

Shown in Figure 4 is an example of an AP Basic program that calls two different subroutines. Note how program execution returns from each called subroutines.

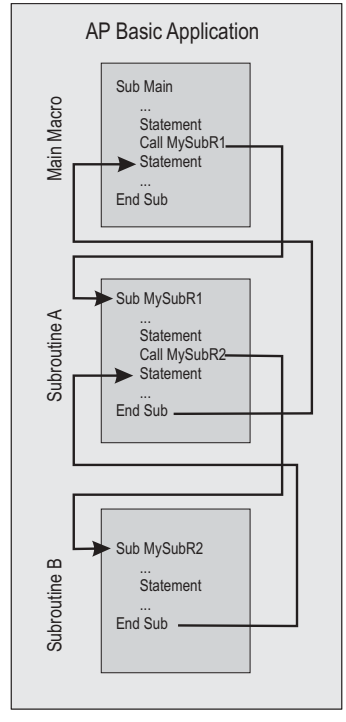


Figure 4.

Calling Subroutines from Other Modules

A subroutine or function can also be called from another macro or code module. It is possible to call subroutines in other macros from anywhere in your program.

To call a subroutine or function in another macro, also known as another code module, you must include a reference to the code module in your macro.

You make the reference to the code module with the '#uses statement. The '#uses statement has the following syntax.

```
'#uses "MODULENAME.APB"
```

An alternative statement to call a subroutine or function in another macro that is also compatible with Microsoft Visual Basic is '$Include:

The '$Include: statement has the following syntax.

```
'$Include: "MODULENAME.APB"
```

There are several important steps you must follow to use the '#uses statement correctly.

- Make sure to include the "' " character in front of the "# " character.
- Add the '#uses statement on the first line of your program
- Include the path to the code module you want to include within the quotes if the code module exists in another directory.

Note that the "' " character is normally used to add comments to your code. It is needed here since the '#uses statement is not a normal AP Basic command and is not compatible with Visual Basic, which uses another form of include.

When you add the '#uses statement to your macro, all of the subroutines and functions of the code module are available to your macro. You call these included subroutines just as you would a normal subroutine.

The following line of code would include all of the subroutines and functions of MYDEMO.APB in your program.

```
'#uses "C:\APWIN\DEVELOPMENT\MYDEMO.APB"

-or-

'$Include: "C:\APWIN\DEVELOPMENT\MYDEMO.APB"
```

One reason for including subroutines and functions from other code modules is that you can create a library of commonly used subroutines. Once you have a library, any program that wants to use a library subroutine just needs to include the appropriate '#uses statement.

To learn more about including subroutines from code modules in your program, refer to the online help.

Objects, Methods, and Properties

In this section we shift from an introduction to subroutines and present some of the more conceptual ideas behind Visual Basic. Much of this conceptual framework centers around how data is represented. For those of you who are new to object-oriented programming, or are new to programming in general, these ideas may seem strange and even confusing. Fortunately, it is not necessary for you to master this section to begin developing AP Basic programs. Instead, the concepts introduced here are intended to expose you to some of the vocabulary and ideas which more experienced programmers use when working with Visual Basic.

What Are Objects?

An object is a combination of code and data that can be treated as a unit. An object may be a part of your program or even the entire program. An object may even represent something physical, like the analog generator of an Audio Precision instrument. Almost anything you want to represent in Visual Basic, either real or imaginary, can be expressed as an object.

Some examples of objects available to you in AP Basic are described in the table below.

Example	Description
Dialog Box	A dialog box that reports information to the user or prompts the user for data is an object.
Chart	A chart in Microsoft Excel is an Object
Database	Databases are objects that can contain other objects, like fields and indexes.
DCX-127 Hardware	Audio Precision's DCX-127 is represented in AP Basic as a library of objects that are contained in the AP class AP.Dcx

Objects are used in AP Basic to make your work as a programmer easier. Since objects can represent complex data structures and code, they can simplify your program by allowing you to use them rather than requiring you to write your own code. For example, you could write your own code to create a chart similar to one you might find in Microsoft Excel, but you don't have to. Instead, you can use Excel to create your chart and then you can manipulate it with the properties of the Chart object.

Usually, when you develop programs in AP Basic, you will only need the objects that are already provided as standard pieces of Visual Basic and AP Basic. However, it is also possible to create your own objects to simplify your code. For more information on creating your own objects refer to any of the suggested texts mentioned in the section **Find Out About Visual Basic** on page 7.

There are three things you can do with objects in AP Basic that make them useful.

- You can set the *value* of an object's *property.*
- You can return the *value* of an object's *property.*
- You can use a *method* of the object to perform a task.

In the last few sections of this chapter we will look more closely at how to use properties and methods to change and control objects.

Working With Objects

Objects in AP Basic support *properties*, *methods*, and *events*. The settings and attributes of an object are called its properties, and the subroutines that operate on an object are called its methods. An event is an action, like pressing a key or clicking the mouse, that is recognized by an object. You can write code to control how an object responds to an event.

Properties of an Object

Properties are special attributes of an object. You use properties to control the appearance of an object, its behavior, or both. A property has a value associated with it that can be read to learn about the condition of an object or set to change the object. For example, an object may have an enabled property you set to **True** to activate the object. To turn Channel A of the analog generator on you would use the AP Basic extension command:

```
AP.Gen.ChAOutput = True
```

To turn the generator off, you set the property to **False**. Sometimes, you may need to know the value of a property without wanting to change the property. To determine the value of property without changing it you assign the value of the property to a variable:

```
variable = AP.Gen.ChAOutput
```

You can now test the variable without altering the property. An alternate way to check a property without changing it is to test the property in more complex expression.

```
If AP.Gen.ChAOutput = True Then
   AP.Gen.ChBOutput = True
Else
   AP.Gen.ChBOutput = False
End If
```

Some objects may also require a parameter be specified to determine the value of a specific property. For example, to determine the amplitude of Channel A on the analog generator of System Two you would use the statement:

```
variable = AP.Gen.ChAAmpl ("V")
```

The (“V”) parameter tells AP Basic that you want the answer to be specified in volts.

Objects often have several properties, some of which may be common to more than one object, while other properties are unique to a single object. A specific set of properties and methods are what makes one object different from another object.

Using the Methods of an Object

Methods are another characteristic of objects. When you use a method associated with object you make the object perform a specific task. To call a method, you use the object name and the method name, separated by a period. For example, using AP Basic code you can open a previously saved test using the *OpenTest* method associated to the *File* object in the *AP* class.

```
AP.File.OpenTest "analog THD measurement.at27"
```

An object may have a number of different methods associated with it. An example of using a second method associated with the File object is:

```
AP.File.OpenWfm "ISO 31 tone generator waveform.aas"
```

Like properties, methods are part of what defines an object. They are useful because they allow you to perform specific tasks without having to write the code yourself.

The Object Browser

The instrument control software is filled with objects you can use in your AP Basic code. To help you search through all the available objects to see what might be useful to you, AP Basic provides a special dialog box called the Object Browser. You can open the Object Browser by pressing the Browse Object button on the subroutine Editor panel. Figure 5 shows what the Object Browser looks like:

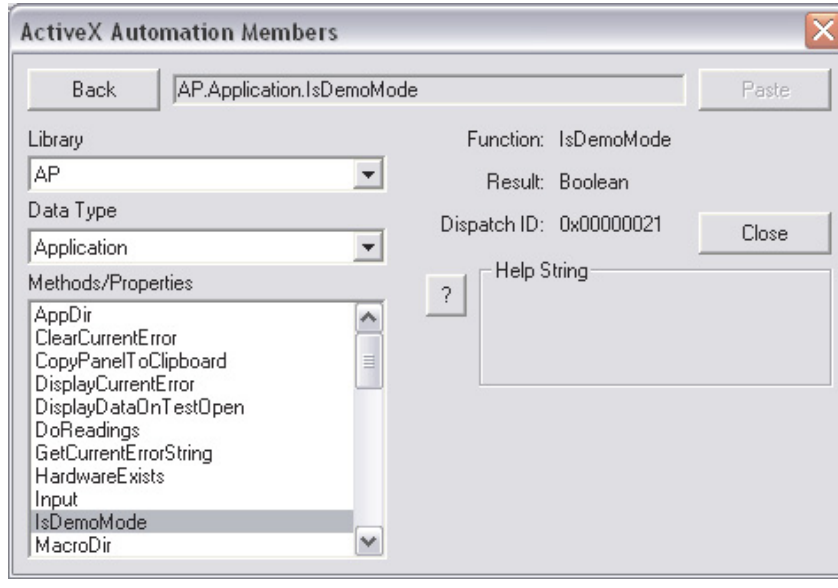


Figure 5.

The Object Browser is a source of useful information about the objects and the code in your application. You can use the Object Browser to learn more about:

- The OLE object libraries available to you.
- The names of all the objects in a given object library.
- The name of all the methods and properties for any object.
- The parameters for a particular method or property.

In addition to the information displayed by the Object Browser, it can also be used to insert an object and its appropriate method or property directly into your code. When you double-click on a method or property in the Object Browser, it will be inserted into your code where the cursor is placed.

All of the methods and properties available in the Object Browser are discussed in greater detail in the Extensions Reference manual for your instrument.

Chapter 3

Writing An AP Basic Macro

Chapter 1 introduced the fundamentals of AP Basic. The theory of macros, objects, methods and properties were discussed in Chapter 2 along with simple examples to familiarize you with the key concepts of Visual Basic programming. Here, in chapter 3, these concepts are applied to create an AP Basic macro.

A complete macro is written with a specific structure and uses keywords and commands to accomplish tasks. Using a simple macro as an example, we will examine what pieces are necessary in an AP Basic macro. Some of the key topics discussed include:

- Using Learn Mode to enter commands directly into your code.
- Macro structure.
- Adding comments to your code.
- Keywords and commands.
- Creating and declaring variables and constants.
- Using conditional statements to control macro flow.

Using Learn Mode

New macros, or additions to existing macros, may be generated by two different techniques. One method, suitable for those with some experience with programming techniques and knowledge of the specific syntax and commands of AP Basic or other forms of Visual Basic, is by typing and modifying text in the Macro Editor. The second method, suitable even for users with little or no experience in programming or AP Basic, is via the LEARN mode (macro Learn Mode menu command) available in the Audio Precision control software. Starting Learn Mode causes each ensuing user mouse click and keyboard entry to write a line of AP Basic code into the Macro Editor. Simple macros may be completely generated in Learn Mode. More sophisticated macros with branching, calling of subroutines or other macros, processing of data results, etc., can have their core created in Learn Mode but will typically require further commands to be added in the Macro Editor.

The Learn Mode Toolbar contains icons to start or stop Learn Mode. When Learn Mode is activated, operator actions including the result of mouse clicks, menu selections, and text or numeric entries into panel fields, will result in lines of AP Basic language code being automatically written into the Macro Editor. The resulting macro can then be run to re-create the series of actions.

Learn Mode is started by clicking on the **Learn** button on the Learn Mode toolbar, or by selecting from the menus **Macro > Learn Mode** or **Utilities > Learn Mode** selections. Once Learn mode has been started, user actions will result in one or more lines of macro code written into the Macro Editor until Learn Mode is halted. If a macro has already been loaded into the Macro Editor, the commands created by Learn Mode will be inserted at the cursor position in the Macro Editor. If no macro has been loaded, the Macro Editor will be opened with a new (blank) macro ready for recording of the Learn Mode commands. To stop Learn Mode, click on the **Stop Learning** button or use the macro Learn Mode or Utilities Learn Mode menu selections again to toggle Learn Mode off. To temporarily suspend the learning of commands, hold down the **Ctrl** and **Shift** keys while clicking the mouse to make changes which will not be learned.

For a Learn Mode example, assume the following list of user actions:

- Click on **Start Learn Mode** button.
- Click on **New Test** button.
- Click on analog generator **OUTPUTS ON/OFF** button..
- Click on analog analyzer Ch A input and select **GenMon** instead of XLR Bal.
- Click on **Page 2** tab.
- Click on the **GO** button (or press **F9**).
- Click on **Stop Learn Mode** button.
- Opening the Macro Editor should show the macro listing as illustrated in Figure 6. This macro will duplicate all the actions above if the Run macro icon is clicked.

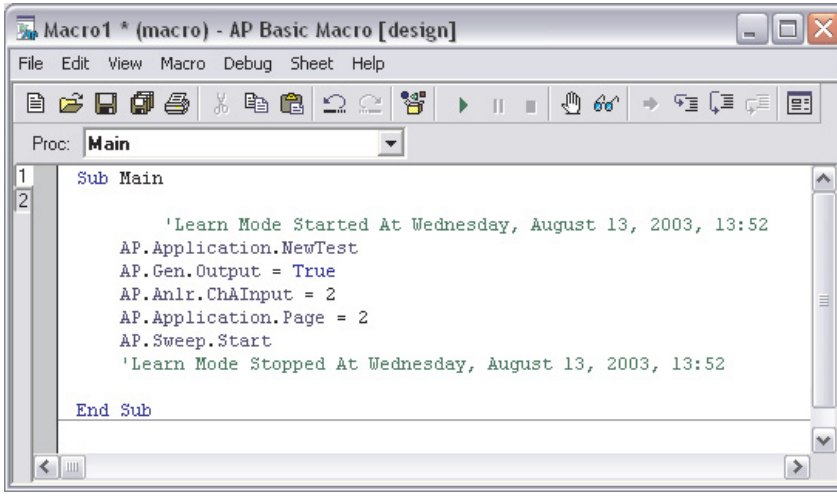


Figure 6.

Example AP Basic macro

This macro is written especially for Audio Precision's APWIN control software.

```
' This macro is designed to assist in creating limit
' files for FFT tests. It Is intended to be executed
' after a test has already been setup and run.
'
' Functionally, this macro will take the results of
' a sweep and limit the low amplitude data points to
' a  specific value. This is particularly useful for
' limit files based on FFT sweeps where the low
' amplitude data is often near the noise floor and
' varies from sweep to sweep.
'
' Algorithmically, the macro operates by
' transferring the sweep data into an array in APWIN
' Basic. This array is scaled from linear units into
' decibels. Each data point in the array is tested
' against a specific limit and if the data is above
' the limit it is left untouched. If it is equal to ' or
below the limit, it is forced equal to the
' limit. Once all the data has been processed it is
' transferred back to APWIN and redisplayed. A limit '
file can then created from this data.


Const Ch1_limit = -110   ' units for limit are in dB
Const Ch2_limit = -110   ' units for limit are in dB
```

```
Sub Main
   Call scale_low_amplitudes
End Sub


Sub scale_low_amplitudes
   size = AP.Sweep.Source1.Steps + 1 ' determine
      number _ of elements in data arrays
   data1 = AP.Data.XferToArray(0, 1)
   data2 = AP.Data.XferToArray(0, 2)
   For i = 0 To size     ' convert data to dB format
      data1(i) = TodB(data1(i))
      data2(i) = TodB(data2(i))
   Next i


   For i = 0 To size ' limit minimum values to -110
   dB
      If data1(i) < Ch1_limit Then
         data1(i) = Ch1_limit
      End If
      If data2(i) < Ch2_limit Then
         data2(i) = Ch2_limit
      End If
   Next i


   For i = 0 To size     ' convert data back from dB
      data1(i) = ToExp(data1(i))
      data2(i) = ToExp(data2(i))
   Next i


   For i = 0 To size      ' write data back to AP
      AP.Data.Value(0,1,i) = data1(i)
      AP.Data.Value(0,2,i) = data2(i)
   Next i
   AP.Data.UpdateDisplay(0)'Show updated results on
   graph
End Sub


Function TodB(x)
   TodB = 20*Log10(x)
End Function 'TodB


Function ToExp(x)
```

```
   ToExp = Exp10(x/20)
End Function 'ToExp
```

Macro Structure

AP Basic macros can be broken down into three main sections:

- a header section.
- the Main subroutine.
- additional subroutines and functions.

The header section of a macro can contain several different parts. Any variables, constants, arrays, and other data types that must be accessible to other code modules should be declared in the header section. The amount of macro code in the header section can vary significantly depending on whether the macro is self contained, or includes other code modules and public variables. You will learn more about how and where to define variables later in this chapter.

A second and often neglected use of the header section is for comments. A good macro header should have a few sentences that identify who wrote the macro, when it was written, what the macro does, and maybe a few words about how it works. Taking the time to add comments to the header section will help you to quickly identify what your macro does and how it works months or even years later when you need to make a change. A more thorough discussion of how and when to use comments is covered in the next section.

Experienced Visual Basic programmers may recognize that it isn't strictly necessary to have a header section for a macro. If you have developed a very simple macro that doesn't use public variables or include other code modules, it is possible to have the first line of your macro begin with the Sub Main declaration. While this minimalist approach will work, it tends to lead to code that is poorly commented and should be avoided.

The Sub Main subroutine was introduced in chapter 1. Its purpose is to identify where macro execution begins and every AP Basic macro must include a Sub Main subroutine to run. Depending on the complexity of your macro, you may only need this one subroutine. More typically, however, the Main subroutine is used as the "top" level of the macro from which other subroutines and functions are called.

Subroutines can be listed in your macro in any order you choose. Consider placing the Sub Main subroutine as the first subroutine in your macro to help others quickly identify where the macro starts. Also, if you are using the main subroutine as the "top" level of your macro, placing it at the start of the macro code will help others to quickly identify the how your macro flows through the various subroutines and functions.

After the Sub Main subroutine, you should place the additional subroutines and functions used in your macro. Again, there are some tricks you can use to help keep your macro as understandable as possible. Structure the subroutines and functions so that they roughly follow the same order as they are used. In complex macros where the same subroutines may be called several different times it may not be possible to follow this rigorously. Your goal in structuring your code should be to keep it as simple and easy to understand as you can make it.

Commenting Code

Properly commented code is an essential part of good programming technique. Code which is not properly documented can be hard to read and difficult to modify. In this section we look briefly at some of the reasons to comment your code as well as some useful guidelines.

One of the biggest temptations to resist when developing code is neglecting to take the time to comment a subroutine you just developed for fear you will loose your train of thought or fall behind schedule. This is usually a mistake. Very few programmers possess the discipline to return to their code when it is finished and add the proper comments. Even worse, after you've been away from your code for a while, it may be difficult to remember how everything works. You may not even remember the reasons why you chose one particular way to implement your code over another.

There are several good reasons to add comments to your code. Among the most compelling are:

- Properly commented code will enable you to quickly identify what a macro does without having to read through the code.
- Comments can help to identify what types of arguments and what ranges of values can be passed to a macro. This will help you to determine where your code can be re-used.
- Comments are the best chance another programmer has for understanding your code. Code which is not commented or commented poorly is often overlooked by other programmers regardless of how well the code may work. If someone else can't easily understand how your code works, they won't use it.

Some of the goals you should work towards when commenting code include:

- Include general comments about a macro that allows other macros to quickly and easily identify what the macros does.

- Identify what input arguments your macro accepts and what outputs it produces. You should also identify any non-local variables that are used or changed.
- Avoid comments that explain what each line of code does. Anyone who understands AP Basic will be able to tell that. What programmers want to see are comments on why your code works the way it does. For example, a `For...Next` loop that counts from one to the number of data points minus one doesn't need a comment saying how many points are counted. What is needed are comments saying why you count up to the number of data points minus one and not all the data points.

Commenting code may seem like an added burden that will slow down code development, but any experienced programmer will tell you that well documented code goes a long way towards developing bug free and re-usable code.

Keywords and Commands

At the beginning of this chapter there is an example of an AP Basic macro. If you study this macro, you will notice that there are several keywords and commands that are used to tell AP Basic what to do. For example, notice the `If...Then` command used at several points in the code. This command, and others like it, are easily identified in the editor by the different color text. The Macro Editor automatically changes the color of keywords and commands as they are entered. You'll find this coloring scheme makes it much easier to read the code and identify the keywords and commands that control macro operation.

A careful observer may have also noticed that none of the variable or constant names are the same as any of the keywords or commands. This is because keywords are reserved in AP Basic. If you try to create a variable with the name *end*, AP Basic will recognize *end* as one of its keywords. When you try to run a macro with a variable named *end* AP Basic will refuse to continue and issue an error message.

For an overview of the different keywords available in AP Basic, select the AP Basic Language option under the Help menu in your Audio Precision control software.

AP Basic offers a large number of keywords and commands to provide you flexibility in creating macros. In the next few sections we will study more closely how to use these to create your own AP Basic macros.

Using Variables and Constants

As you develop an AP Basic macro, you will often need to store information in your macro, even if only temporarily. For example, you might need to calculate a running sum of data and you want to be able to store this value

while your code loops through all the data. AP Basic, like other programming languages, uses *variables* for storing information. Depending on the type of variables you use, the information stored in a variable may only be available during the short time in which your macro uses it, or the information may be preserved during the entire time the macro is executed.

A variable stores information which may change as your macro is run. In order to use variables, Visual Basic must know something about the type of data the variable will store, known as the *data type*. It must also have a name, or label it uses to refer to the value the variable contains.

A *constant* is similar to a variable except its value does not change as the macro is executed. You use constants to simplify your code and make it easier to read. Like variables, constants have specific names and data types.

Declaring Variables

Before AP Basic can use a variable, that variable must first be *declared*. Declaring a variable means that AP Basic reserves a location in memory to store information that is assigned to the variable. The amount of memory reserved depends on the data type used.

Variables can be declared in one of two ways, either *explicitly* or *implicitly*. An explicitly declared variable is created by a specific line of code that identifies the variable name and, optionally, its data type. An implicitly declared variable is not specifically identified in a separate line of code, but is used just as if it had been explicitly declared.

There a several statements used in AP Basic to declare variables. The following table briefly describes these statements and when they should be used.

Declaration Statement	Description
`Dim`	Used to declare variables within subroutines or functions that have local scope.
`Static`	Used to declare variables within a subroutine or function block that will retain its value over multiple sub or function calls.
`Public`	Used to declare variables shared by all files in a project. A project may contain several different files.
`Private`	Used to declare variables available only to the current file (module).

Variables declared with the `Dim` statement follow the general form:

```
Dim VariableName As DataType
```

All other variable types are declared in the same way, by adding the declaration statement before the variable name.

```
Public VariableName As DataType
Private VariableName As DataType
Static VariableName As DataType
```

Note that any variables declared as `Public` should be placed at the beginning of your (file) macro before any sub or function macros. Public variables cannot be declared within a subroutine or function block..

Scope of Variables

Variables can be created that are accessible to all subroutines or function blocks in a macro, or they can be restricted to use only in a specific sub or function. How visible a variable is to different macros is known as the scope of the variable. There are three levels of scope:

- Local.
- Module (file) level.
- Public.

Local variables have the narrowest scope. They are only visible to the sub or function where they are declared and used. This means you can have several variables in your macro, each with the same name, as long as they are declared locally in separate sub and functions..

To ensure a variable is local, declare it either implicitly or explicitly inside a sub or function. Here is an example subroutine with three locally declared variables, two of which are declared explicitly (A1 and A2) and one of which is declared implicitly (A3):

```
Sub DoSomething
   Dim A1 As String
   Static A2 As Integer
   A3 = 4.0
   ...
End Sub
```

Local variables are useful when you need to temporarily store information in a sub or function. A local variable declared implicitly or with the `Dim` statement will be removed from memory when the sub or function is finished executing. A `Static` variable will remain in memory and retains its value next time the function or sub is called. By definition, all local variables are private to the macro in which they are used.

Module level variables have a much broader scope than local variables. A module level variable is visible to all sub or functions in the module (remember, a module is the same as a .apb file, and you can link together several different code modules with the `'#uses` command discussed in the previous chapter).

To create a module level variable it must be declared outside of any sub or function. Typically, you should place these in the header section of your module and declare them `public` or `private`..

The primary advantage of module-level variables is that they can be used to easily share information between different subs or functions. When one module assigns a `public` value to a module-level variable, a second module can access and use that same information.

Public variables have the broadest scope and are visible to all sub and functions in an application, regardless of the module that contains them. They are declared using the `Public` statement and should be placed at the top of a module prior to the first sub or function. Here is a simple example of declaring and using a `Public` variable.

```
Public Y As Integer
   Sub Main
   Y = 1
   Y = Y + 10
   ...
End Sub
```

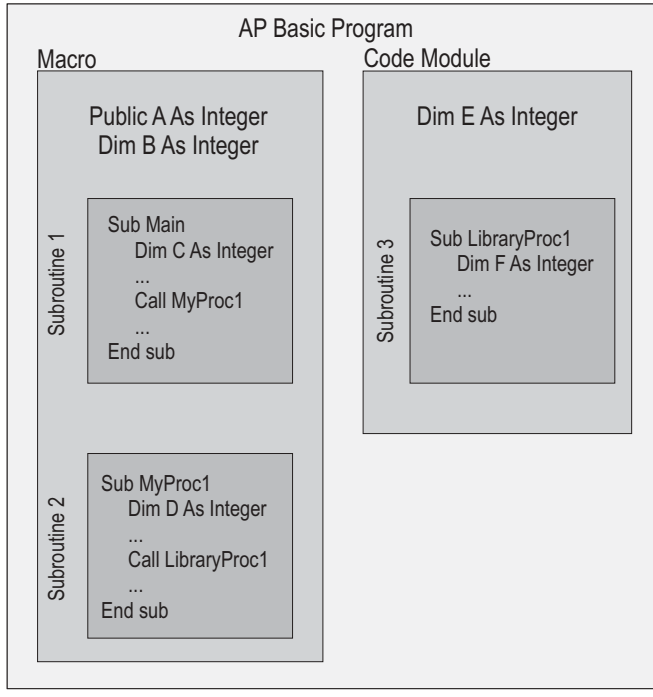


Figure 7.

Macro	Variable visible to macro
1	A, B, C
2	A, B, D
3	A, E, F

Figure 7 shows how the scope and visibility of variables change depending on how and where they are declared.

When AP Basic is executing code, it evaluates variables starting from the narrowest scope to the broadest. Therefore, if your code contains a local variable, a module level variable, and a `public` variable each with the same name, AP Basic will look first for a local variable with the desired name, then for the module level variable, and finally, it will check for a `public` variable.

Data Types

When you declare a variable, you can optionally supply a data type. A data type is a property that identifies what type of data is stored in a variable. The data type specifies two things:

- the type of data (i.e. text, numeric, object)
- the range of values for the data

The following table describes a few of the more common data types available in AP Basic.

Data Type	Storage Size	Range
Integer	2 bytes	–32,768 to 32,767
Single	4 bytes	±3.4 E38 to ±1.4 E-45
String	1 byte per character	0 to approximately 65,500 characters
Boolean	2 bytes	True or False
Variant	16 bytes + 1 byte	depends on data type assumed for each character.

You can learn more about all of the available data types in the online help.

The Variant Data Type

The variant data type is a special data type. By default, any variable that is not explicitly assigned a data type will be assumed to be variant. It is the most flexible data type available in AP Basic since it can assume the value of any other data type. The particular data type a variant assumes depends on how the variable is used. For example, a variable with the variant data type can be assigned an integer value at the start of a macro, and then be reassigned to a string value later in the code. It changes data types depending on how it is used. Consider the following example:

```
Dim FFTSize       ' Variant data type by default
FFTSize = "1024"    ' FFTSize is a string data type
FFTSize = FFTSize * 8
        ' FFTSize changes to a numeric
             ' data type equal to 8192
FFTSize = "Big" & FFTSize
             ' FFTSize is now a string
             ' again containing "Big8192"
```

Constants

A *constant* is a name you choose to replace a value used in your macro. They are used to help make code both easier to read and to modify.

For example, suppose you need to use the value of Pi = 3.145926535 at several different places in your code. You could type in the value of Pi each time

you need it, but this takes time and is prone to error. Instead, using a constant with the name Pi will be faster and easier to read. Later in your code if you determine you wanted to use 2*Pi instead, you only need to change the value of the constant.

You declare constants with the `Const` statement:

```
Const name = value
```

Here is how to use Pi as a constant:

```
Const Pi = 3.145926535
```

You don't need to declare the data type for a constant because AP Basic simply determines the data type based on its value. For the example shown above, Pi is assigned the *double* data type.

Controlling Macro Flow

In this section you will learn how to write macros that can test conditions and run certain branches of code depending upon the results. The AP Basic commands that make decisions and alter code flow are called control structures. A second class of commands known as loop structures can be used to execute the same section of code multiple times.

Earlier, when introducing macros it was said that code is executed in a macro from top to bottom, one line at a time. Although simple macros can be written using such linear flow, much of the power and utility of AP Basic comes from its ability to use control structures to change the order in which code is run.

The diagram in Figure 8 illustrates the three most common types of macro control flow.

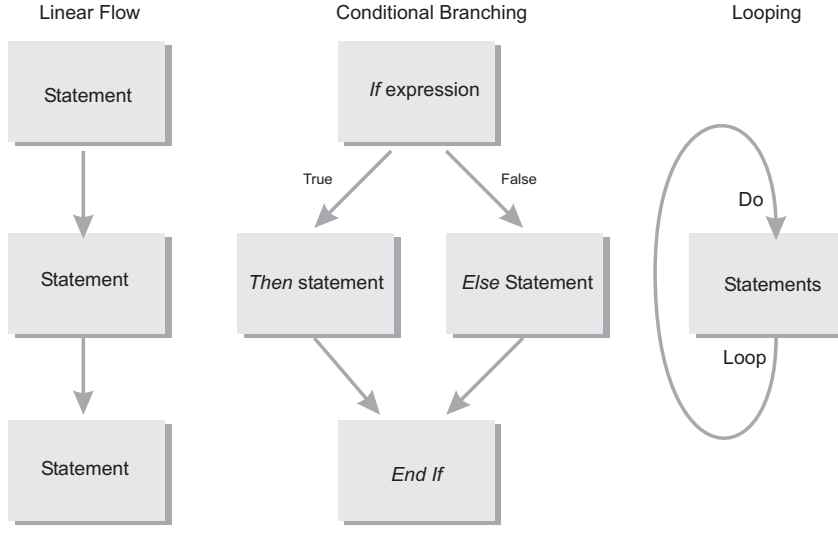


Figure 8.

Control Structures

If...Then

The *If...Then* structure is used to run a section of code depending on the evaluation of a test expression. The test expression must be either true or false. When the expression is true, the section of code inside the If...Then structure is run. If the expression is false, the code is skipped.

You can use either a single-line syntax or a multiple-line syntax.

```
If FFTSize 2048 Then MsgBox "Use a larger FFT Size"
```

- OR -

```
If FFTSize < 2048 Then
   MsgBox "Use a larger FFT Size"
End If
```

Notice that the multiple-line syntax uses the `End If` statement to identify where the code section ends. If you want to run more than one line of code when the condition is true, you must use the multiple-line syntax.

```
If FFTSize < 2048 Then
   FFTSize = 2048
   MsgBox "FFT Size has been increased to 2048"
End If
```

If...Then...Else

This is a more flexible form of the If...Then structure. It allows you define more than one section of code, one of which is always run.

```
If Age 18 Then
   MsgBox "You are too young to vote."
Else
   MsgBox "You are old enough to vote."
End If
```

You can add the `Else If` statement within the `If..Then` structure for even more flexibility.

```
If Season = "Summer" Then
   Temperature = "hot"
Else If Season = "Spring" Or "Fall" Then
   Temperature = "mild"
Else
   Temperature = "cold"
End If
```

Notice that last possible season, “Winter” was not tested with an `Else If` statement. If the season is neither summer, spring, or fall, then it must be winter. It is possible to use the `Else If` statement to test for winter, but you would get the same result.

Select Case

AP Basic provides the `Select Case` statement as an alternative to `If...Then...ElseIf`. The select case statements searches for matching values to an expression instead of testing whether the expression is true or false. Often, it is used to make code more efficient and readable.

```
Select Case Percentile
 Case Is > 50
  MsgBox "Above the 50th percentile"
 Case 50
```

```
  MsgBox "perfectly average"
 Case Else
  MsgBox "Below the 50th percentile"
End Select
```

Notice the use of the `Is` operator to compare a range of values to the initial expression.

The first line of code in a select case statement identifies the expression to be evaluated. For the example just given, the expression is Percentile. The select case statement can be used to evaluate only one expression, unlike the `If...Then...Else` structure which can test several different, even unrelated, expressions.

Loop Structures

For...Next

The `For...Next` structure is used to loop through a section of code a specific number of times. It uses a variable to count the number of times the loop has been run. Depending on how you want the code to run, the variable is incremented or decremented on each loop through the code. Execution stops when the variable reaches a predetermined value.

```
For y = 1 To 10
 MsgBox "The count is currently " & CStr (y)
Next y
```

In this example, y is the count variable. It is initialized to 1 at the start of the loop and is incremented on each pass. A message box indicates the value of the y. When y is equal to 10 a final message is given and the loop terminates.

You can make the `For...Next` structure more flexible by counting either up or down and by using a variable step size.

```
For i = 16 To 4 Step -2
 MsgBox "The count is currently " & CStr (i)
Next i
```

This example will count down from 16 to 4 by steps of two.

Do...Loop

The `Do...Loop` structure is used to count an indeterminate number of times. Instead of a count variable, it uses a test expression to determine when execution should stop. In this way, a `Do...Loop` structure will run until the expression is satisfied.

```
Sub IncrementByTwo (x)
   Dim LimitReached As Boolean
   LimitReached = False
   loopCount = 0
   Do Until LimitReached
      x = x + 2
      If x > 100 Then
         MsgBox "The limit was reached in " & _
            CStr(loopCount) & " loops"
         LimitReached = True
      Else
         loopCount = loopCount + 1
      End If
   Loop
End Sub
```

This subroutine accepts an unknown input x from the calling macro. It then increments the value of x by two until x is greater than 100. When the test condition is satisfied the boolean expression `LimitReached` is changed from false to true and a message is given reporting the number of times the loop was run.

An alternate way to use the `Do...Loop` structure is use the `Do While` clause instead of the `Do Until` clause. If you use the `Until` clause, the loop runs as long as the expression is false. When you use the `While` clause the loop runs as long as the expression is true. Its important that the code in a `Do...Loop` structure provides a means to alter the test expression. If the test expression can't change, AP Basic will not be able to exit the loop.

User Notes

Chapter 4

Testing and Debugging

Once you have written an AP Basic application, you need to determine if your application runs properly. This is part of testing your code. If it does not run correctly, you need a means to fix these errors, also known as debugging your code. AP Basic cannot diagnose or fix errors for you, but it does provide a number of tools to help you analyze how your code operates.

AP Basic uses an Interactive Design Environment (IDE) to assist in detecting and fixing errors in your program. In this environment it is possible to stop your code at any point during execution and display the state of variables and properties. You can also step through your code one line at a time while watching how settings change. The ability to interact with your code as it is executing is a powerful debugging tool.

Unfortunately, there are no magic tricks to debugging, and there are no steps that always catch errors. Debugging is really part of a process to help you better understand how your code is operating. Using the debugging tools provided in the Interactive Design Environment it is possible to more easily identify and correct the problems that keep your application from running properly.

Types of Programming Errors

Before exploring how to test and debug code, consider the kinds of errors you might encounter.

- Syntax errors occur when code is improperly written. For example, incorrectly typing a keyword, using incorrect punctuation, and omitting key words are all forms of syntax errors. AP Basic will detect and flag these errors before the code is run.
- Run-time errors result when a section of code is impossible to execute. A common example you may have encountered before is a divide by zero error. These types of errors cannot be detected until the code is executed. When AP Basic encounters a run-time error, program execution is halted.
- Logic errors are the most common and can be one of the most difficult types of errors to fix. A logic error occurs when code doesn't operate the way it was intended. Even though the code may be syntactically correct and will run without errors, it may not produce the results you expect.

AP Basic cannot detect logic errors since it can't know how your program should work. It does, however, provide a number of tools to help you diagnose logic errors.

As you first develop your code, you're likely to create a number of syntax errors. These are easy to detect since AP Basic will point them out to you by highlighting the affected line in red and placing the cursor close to the suspected error when you run the macro. As you become more proficient in AP Basic, you will tend to make fewer syntax errors.

Once your program is syntactically correct, you can execute it. At this point, you may or may not encounter run-time errors. These errors often occur only for certain types of input data, so you may or may not see them the first time your program runs. In fact, you may have to run your code several different times and with several different sets of data before you see a run-time error.

Lastly, you may notice logic errors when your program runs but behaves differently than you expected. Any of these three types of errors will require you to review your code, identify the source of the bug, and re-write your code to fix the error.

Debugging Tools on the Toolbar

The Macro Editor has a number of buttons used for debugging code. These buttons are found near the top of the Macro Editor panel.

The following table describes the function of each button:

Debugging Tool	Purpose
 Breakpoint	Used to mark a line in the code where Visual Basic will suspend execution.
 Quick Watch	Displays the value of the expression under the cursor while in break mode.
 Step Into	Executes the next line of code in the application and steps into subroutines.
 Step Over	Executes the next line of code in the application without stepping into subroutines.
 Step Out	Steps out of the current subroutine or function.

These debugging tools are designed to help you observe the behavior of your code and enable you to diagnose and fix run-time and logic errors. In or-

der to use these tools effectively, you need to understand how they can be utilized during program operation.

Break Mode

Break mode is a special operating mode of AP Basic that allows you to halt program execution and examine the state of variables and expressions in your code. When you enter break mode:

- The Debug window automatically appears in the Macro Editor panel as shown in Figure 10. The Debug window includes several different window panes that provide useful debugging information.
- You are temporarily prevented from editing your code. Since you have actually just suspended execution but not stopped execution, AP Basic does not allow you to add and remove commands from your program.

Once you have entered break mode, the value of all variables and expressions is preserved, so you can check their current state. Depending on whether or not your program is running correctly, you may want to change the value of several variables and expressions as well. In break mode it is possible to interact with program operation in several ways.

While in break mode you can:

- Check the value of variables, expressions, and properties.
- Modify the value of variables and expressions.
- Use the immediate pane in the Debug window to run AP Basic commands not included in your program.
- Step through operation of your code one line or one subroutine at a time.

Accessing Break Mode

AP Basic will enter break mode when any of the following occur:

- Execution reaches a line of code with a breakpoint.
- Execution reaches a Stop statement.
- A line of code generates a run-time error.
- Program execution is started by pressing either the **Step Into, Step Over,** or **Step Out** buttons.

The most common technique for accessing break mode is to add breakpoints to your code. AP Basic will enter break mode and suspend execution on the line of code just before the breakpoint.

To add a breakpoint, move the cursor to the line of code where you want to place a breakpoint and press the toggle breakpoint button. When you set a breakpoint, AP Basic will mark the selected line of code by highlighting the line and adding a dot to the left of it, as shown in Figure 9. To remove a breakpoint, select the desired line of code and press the toggle breakpoint button.

A second way of entering break mode is to add the Stop command to your code. This is most useful when you need to ensure program execution halts at a particular point. Notice, there is an important difference between breakpoints and the Stop command. Breakpoints are lost when you close and reload your program, but Stop statements stay in the code until you remove them.

Regardless of how you entered break mode, you can always resume execution by pressing the run/resume button or by continuing to step through your code.

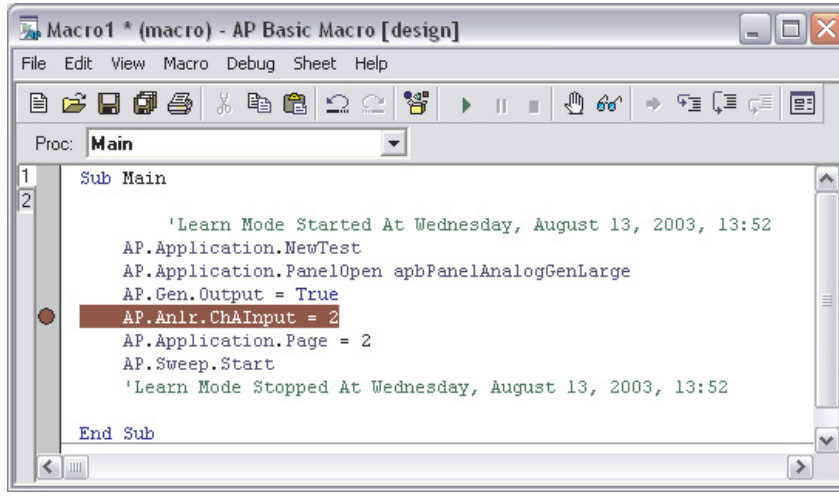


Figure 9.

Stepping Through Code

Once you've identified a potential trouble spot in your code, it is useful to continue executing your code one line at a time. This allows you to see how each line affects the behavior of the application as well as the values of variables and other data. Executing code one line at a time is called *stepping through code*. AP Basic provides three different tools to step through your code.

- Step Into
- Step Over

- Step Out

These three tools operate nearly the same. When you press any of them, AP Basic will execute the next line of code and then return to break mode. They differ in how they execute a line of code that either calls another subroutine or that exists inside of a called subroutine.

For example, if the current line of code to be executed is a call to another subroutine, Step Into will move into that next subroutine. Step Over, on the other hand, will not descend into the called subroutine. Instead, it executes all the commands in the called subroutine and halts immediately after returning to the calling subroutine. This is useful if you are reasonably certain that the bug you're looking for isn't in the called subroutine and you don't want to take the time to step through it.

Step Out will execute all the commands in the current subroutine until it has returned to the calling subroutine. Once it has reached the calling subroutine it halts execution and returns to Break Mode. You should use Step Out if you have stepped through all the code in the current subroutine you are interested in and you want to return to the calling subroutine. Note, if you press Step Out from the Main subroutine, and you have not added any additional breakpoints to your code, the program will run to completion.

Using The Debug Window

In the Debug window, you can monitor the values of expression and variables while stepping through the statements in your code. There are four window panes available in the Debug window, the Immediate, Watch, Stack, and Loaded. Each of these window panes can provide useful debugging information about your program.

You display the debug window by:

- Entering Break Mode. The Debug window is automatically opened when AP Basic enters Break Mode.
- Choosing **View** and then **Always Split** from the menu options available when you right-click the mouse in the main editor window.

This will leave the Debug window visible in the Macro Editor panel as shown in Figure 10.

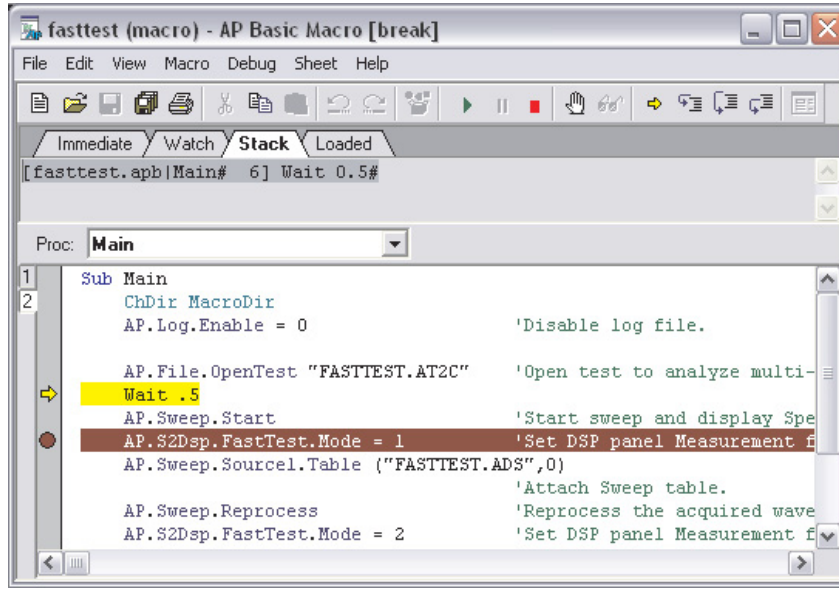


Figure 10.

The *Watch pane* displays information about expressions and variables you tell the control software to monitor as your code is executing. The *Immediate pane* allows you to enter additional AP Basic commands to learn more about your code. Typically, you use the Immediate pane to change the value of a variable or expression. The Stack pane shows you information about what line of code is currently active and what subroutines have been called to reach the current line. Finally, the *Loaded pane* indicates all the .apb files that have been loaded and are being used by the current program.

Additional information about all of the window panes shown in the Debug window is available in the online help.

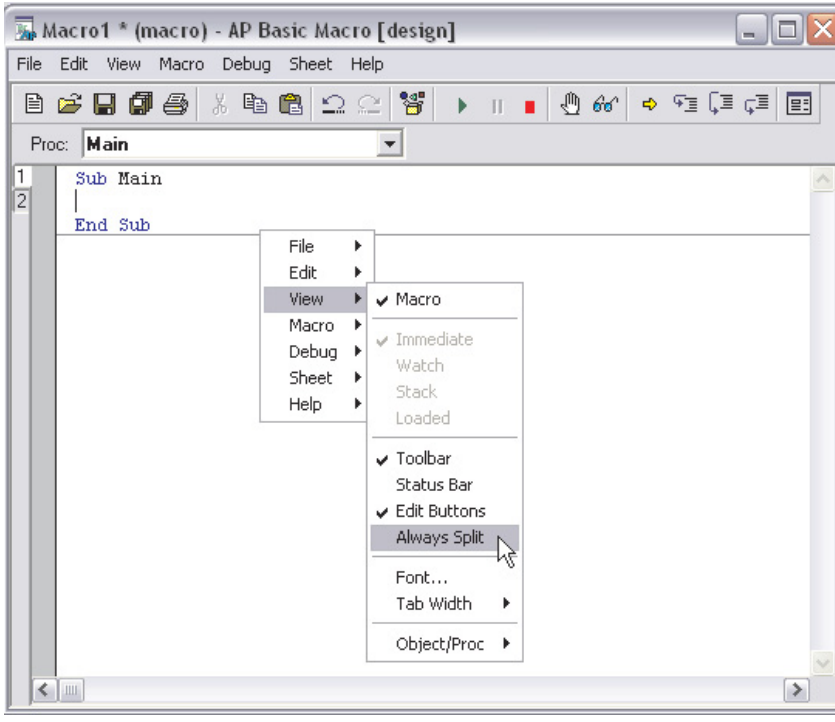


Figure 11.

Normally, the Debug window automatically displays when the macro is run. If you want the Debug wind to be displayed when the macro is not running, simply click the right mouse button and select **View, Always Split** from the menu as shown Figure 11.

Error Handling

In addition to testing and debugging your code, it is valuable to consider the different ways you can develop code to handle errors that occur while your program is running. When a run-time error occurs, AP Basic will usually generate an error message that halts your code. Often, there's nothing the user can do to resume running the application. Other errors might not interrupt execution, but they may cause it to act unpredictably. From a programmers standpoint, it's important to know how to write code that can detect run-time errors and branch to special code that will recover from the errors without halting your program. Adding code to recover from errors is known as *error handling*.

There are several different ways run-time errors can be generated. Earlier, when discussing the different types of errors, it was mentioned that code attempting a divide by zero will generate a run-time error. More generally, a run-time error occurs whenever your code attempts an invalid instruction. For

example, you might have a subroutine that prompts the user to enter the name of a test file to run. If the user enters an invalid name or a name that does not exist, AP Basic will not be able to continue. In this section, we consider different techniques you can use to recover from run-time errors.

AP Basic Error Handling Commands

AP Basic provides a number of commands to allow you to detect and handle run-time errors before they halt your program (a program that abruptly halts operation and won't continue is said to have *crashed*). Intercepting an error is also known as *trapping* an error. You can use the following statements to trap and then respond to run-time errors:

- The **On Error Goto** command can be used to branch in your code when an error is detected. It must be set up before the run-time error occurs.
- The **Err** function returns the number corresponding to the most recent run-time error.
- The **Error** function returns message text corresponding to an error number. Every run-time error has a corresponding error number that identifies it.

The following example uses all three types of error handing commands:

```
Sub Main
   X = 1
   Y = 0
   On Error GoTo ErrorMessage
      Z = X/Y ' create a divide by zero error
      ' At this point the code moves to the _
       ErrorMessage section
   Exit Sub      ' leave the subroutine at this
                 point
 ErrorMessage:
   MsgBox "The most recent error number is " _
   & Err & ". The error message is: " & Error(Err)
   Resume Next   ' return to next line of code
              after the error occurred
End Sub
```

When you run this program, it will generate a message box that says, "The most recent error number is 10061. The error message is: Divide by zero."

Notice that this example has introduced several new programming techniques. The first technique to consider is the use of the **Goto** command. When-

ever the **Goto** command is used, it must refer to a line label in your program. In the preceding example, the line label used in the **Goto** command was "ErrorMessage:" All line labels must follow the standard AP Basic naming conventions and must end with a colon.

The second technique to notice is the use of the line continuation command. This is the underscore character "_", seen at the end of the line beginning with the MsgBox command. The line continuation command tells AP Basic to wrap the next line of code into the current line of code.

Lastly, the **Resume Next** command is used to return from error branching. It allows your program to continue normal operation after handling the error condition.

The process of trapping errors can be summarized as:

- Setting an error trap.
- Writing code to handle to the error.
- Returning to normal program execution.

User Notes

Chapter 5

Creating Custom User Interfaces

Many of the macros you are likely to develop in AP Basic will be designed to assist in automating tests and simplifying complex measurements. One of the most powerful ways to simplify using a macro is to include a custom user interface (UI). You create a custom UI by adding code that will create dialog boxes and custom menus when your macro is executed.

A custom user interface can be very useful when you want to guide a novice user through running a number of different tests. For example, a macro might begin by presenting the user with a custom menu that offers several different tests to run. Different tests can be linked to different menu options depending on the type of measurement needed. The user can only select from the tests available. When a chosen test is complete, the results can be printed out or logged to a file and the macro then returns to the initial custom menu as shown in Figure 12.

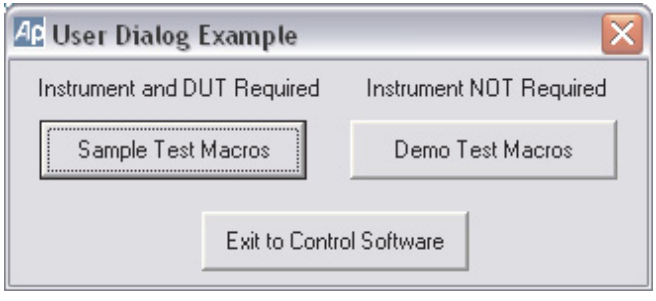


Figure 12.

This section explains how to use dialog boxes and menus to customize the user interface to your macros. The different tasks you can complete with dialog boxes and menus include:

- Getting information from the user. A typical example might include querying the user for their initials which can be logged in the test report.
- Displaying information to the user. Message boxes can be developed indicating how the hardware should be connected or what errors may have occurred while testing.

- Simplifying the interface of the control software with custom menus. With a properly constructed custom interface, a user does not need to be familiar with the subtleties of the control software.

To assist in developing custom dialog boxes and menus, AP Basic includes a User Dialog Editor shown in Figure 13. To access the User Dialog Editor click the Edit UserDialog button in the Macro Editor. This will open a default template for a dialog box. You can select from the menu bar on the left of the dialog box editor to define regions of text in your message box as well as locations for push-button controls or user input. Figure 14 shows the highlighted code for a previously created dialog box. Once highlighted, click the **Name** button to edit the dialog box.

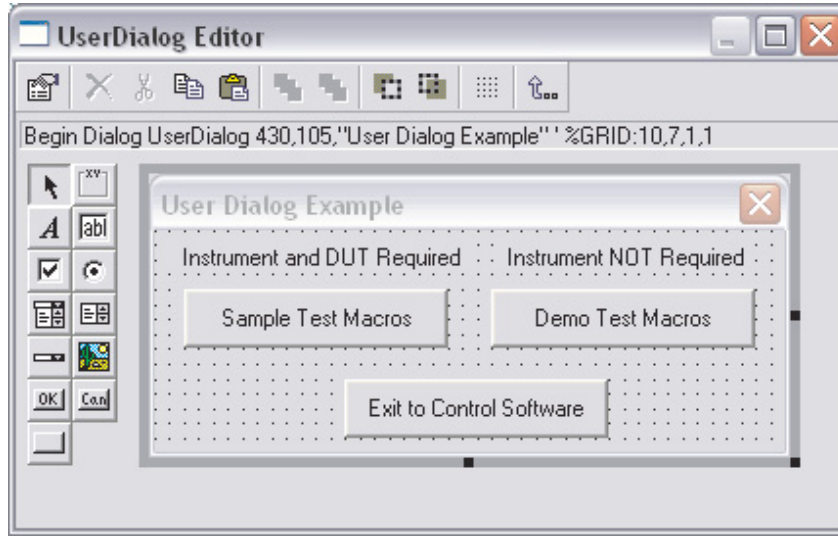


Figure 13.

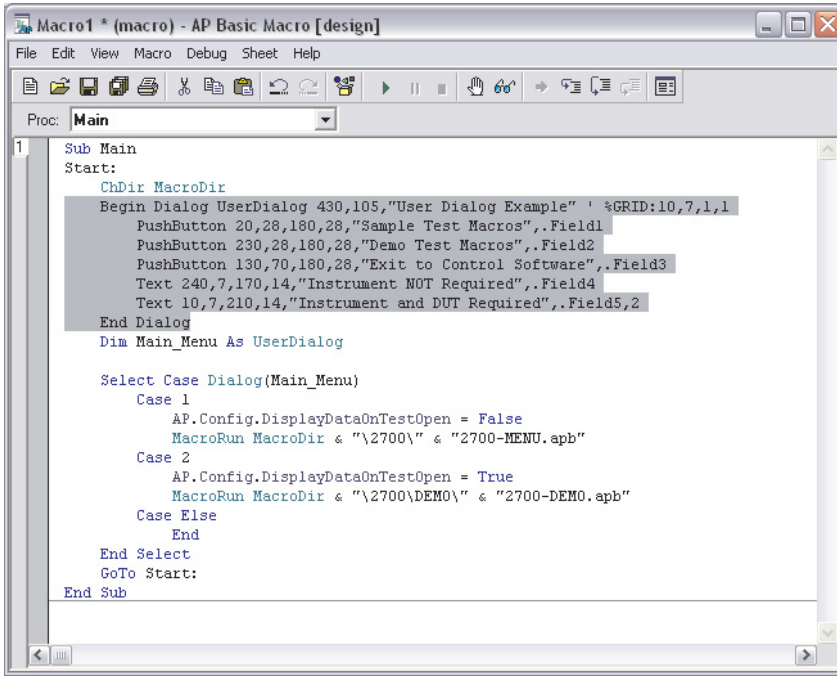


Figure 14.

An example of implementing a custom user interface is shown below. Notice that when the macro is run, the code remains in a loop waiting for the user to select a menu option. When a particular option is selected, the **Macro Run** command is used to launch a second macro that executes the desired test. When complete, the macro will close and return to the main loop.

```
Sub Main
Start:
   ChDir MacroDir
   Begin Dialog UserDialog 430,105,"User Dialog Example"
         ' %GRID:10,7,1,1
      PushButton 20,28,180,28,"Sample Test Macros",.Field1
      PushButton 230,28,180,28,"Demo Test Macros",.Field2
      PushButton 130,70,180,28,"Exit to Control_
         Software",.Field3
      Text 240,7,170,14,"Instrument NOT Required",.Field4
      Text 10,7,210,14,"Instrument and DUT_
      Required",.Field5,2
   End Dialog
   Dim Main_Menu As UserDialog
```

```
   Select Case Dialog(Main_Menu)
      Case 1
         AP.Config.DisplayDataOnTestOpen = False
         MacroRun MacroDir & "\2700\" & "2700-MENU.apb"
      Case 2
         AP.Config.DisplayDataOnTestOpen = True
         MacroRun MacroDir & "\2700\DEMO\" & _
            "2700-DEMO.apb"
      Case Else
         End
   End Select
   GoTo Start:
End Sub
```

Chapter 6

Language Reference

Introduction

Groups

Declaration	#Reference, #Uses, Attribute, Class Module, Code Module, Const, Declare, Deftype, Dim, Enum...End Enum, Function...End Function, Object Module, Option, Private, Property...End Property, Public, ReDim, Static, Sub...End Sub, Type...End Type, WithEvents.
Assignment	Erase, Let, LSet, RSet, Set.
Flow Control	Call, CallByName, Do...Loop, End, Exit, For...Next, For Each...Next, GoTo, If...ElseIf...Else...EndIf, MacroDir, MacroRun, MacroRunThis, Select Case...End Case, Stop, While...Wend,
Error Handling	Err, Error, On Error, Resume.
Conversion	Array, CBool, CByte, CCur, CDate, CDbl, CInt, CLng, CSng, CStr, CVar, CVDate, CVErr, Val.
Variable Info	IsArray, IsDate, IsEmpty, IsError, IsMissing, IsNull, IsNumeric, IsObject, LBound, TypeName, UBound, VarType.
Math	Abs, Atn, Cos, dBToPowerRatio, dBToVoltageRatio, Exp, Exp10, Fix, Int, Log, Log10, Pow, PowerRatioTodB, Randomize, Rnd, Round, Sgn, Sin, Sqr, Tan, VoltageRatioTodB.
String	Asc, AscB, AscW, Chr, ChrB, ChrW, Format, Hex, InStr, InStrB, InStrRev, LCase, Left, LeftB, Len, LenB, LTrim, Mid, MidB, Oct, Replace, Right, RightB, RTrim, Space, String, Str, StrComp, StrReverse, StrConv, Trim, UCase.
Object	CreateObject, GetObject, Me, With...End With.
Time/Date	Date, DateAdd, DateDiff, DatePart, DateSerial, DateValue, Day, Hour, Minute, Month, MonthName, Now, Second, Time, Timer, TimeSerial, TimeValue, Weekday, WeekdayName, Year.
File	ChDir, ChDrive, Close, CurDir, Dir, EOF, FileAttr, FileCopy, FileDateTime, FileLen, FreeFile, Get, GetAttr, Input, Input, Kill, Line Input, Loc, Lock, LOF,

	MkDir, Name, Open, Print, Put, Reset, RmDir, Seek, Seek, SetAttr, Unlock, Write.
User Input	Dialog, GetFilePath, InputBox, MsgBox.
User Dialog	Begin Dialog...End Dialog, CancelButton, CheckBox, ComboBox, DropListBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, PushButton, Text, TextBox.
Dialog Function	Dialog Func, DlgControlId, DlgCount, DlgEnable, DlgEnd, DlgFocus, DlgListBoxArray, DlgName, DlgNumber, DlgSetPicture, DlgText, DlgType, DlgValue, DlgVisible.
DDE	DDEExecute, DDEInitiate, DDEPoke, DDERequest, DDETerminate, DDETerminateAll.
Settings:	DeleteSetting, GetAllSettings, GetSetting, SaveSetting
Miscellaneous	AppActivate, Attribute, Beep, CallersLine, Choose, Clipboard, Command, Debug.Print, DoEvents, Environ, IIf, MacroDir, QBColor, Rem, RGB, SendKeys, Shell, Wait, WaitAndDoEvents.
Operator	Operators: +, -, ^, *, /, \, Mod, +, -, &, =, <>, <, >, <=, >=, Like, Not, And, Or, Xor, Eqv, Imp, Is.

Operators

Syntax

```
^ Not * / \ Mod + - & < <= > >= = <> Is And Or Xor
Eqv Imp
```

Description

These operators are available for numbers n1 and n2 or strings s1 and s2. If any value in an expression is *Null* then the expressions value is *Null*. The order of operator evaluation is controlled by operator *precedence*.

Operator	Description
-n1	Negate *n1*.
n1 ^ n2	Raise *n1* to the power of *n2*.
*n1 * n2*	Multiply *n1* by *n2*.
n1 / n2	Divide *n1* by *n2*.
n1 \ n2	Divide the integer value of *n1* by the integer value of *n2*.
n1 Mod n2	Remainder of the integer value of *n1* after dividing by the integer value of *n2*.
n1 + n2	Add *n1* to *n2*.
s1 + s2	Concatenate s1 with *s2*.
n1 - n2	Difference of *n1* and *n2*.
s1 & s2	Concatenate s1 with *s2*.
n1 < n2	Return *True* if *n1* is less than *n2*.
n1 <= n2	Return *True* if *n1* is less than or equal to *n2*.
n1 > n2	Return *True* if *n1* is greater than *n2*.
n1 >= n2	Return *True* if *n1* is greater than or equal to *n2*.
n1 = n2	Return *True* if *n1* is equal to *n2*.
n1 <> n2	Return *True* if *n1* is not equal to *n2*.
s1 < s2	Return *True* if s1 is less than *s2*.
s1 <= s2	Return *True* if s1 is less than or equal to *s2*.
s1 > s2	Return *True* if s1 is greater than *s2*. \
s1 >= s2	Return *True* if s1 is greater than or equal to *s2*.
s1 = s2	Return *True* if s1 is equal to *s2*.
s1 <> s2	Return *True* if s1 is not equal to *s2*.
Not n1	Bitwise invert the integer value of *n1*. Only Not *True* is *False*.
n1 And n2	Bitwise and the integer value of *n1* with the integer value *n2*.
n1 Or n2	Bitwise or the integer value of *n1* with the integer value *n2*.
n1 Xor n2	Bitwise exclusive-or the integer value of *n1* with the integer value *n2*.
n1 Eqv n2	Bitwise equivalence the integer value of *n1* with the integer value *n2* (same as Not (n1 Xor n2)).
n1 Imp n2	Bitwise implicate the integer value of *n1* with the integer value *n2* (same as (Not n1) Or n2).

Example

```
Sub Main
    N1 = 10
    N2 = 3
    S1$ = "asdfg"
    S2$ = "hjkl"
    Debug.Print -N1        '-10
```

```
    Debug.Print N1 ^ N2     ' 1000
    Debug.Print Not N1      '-11
    Debug.Print N1 * N2     ' 30
    Debug.Print N1 / N2     ' 3.3333333333333
    Debug.Print N1 \ N2     ' 3
    Debug.Print N1 Mod N2   ' 1
    Debug.Print N1 + N2     ' 13
    Debug.Print S1$ + S2$   '"asdfghjkl"
    Debug.Print N1 - N2     ' 7
    Debug.Print N1 & N2     '"103"
    Debug.Print N1 < N2     'False
    Debug.Print N1 <= N2    'False
    Debug.Print N1 > N2     'True
    Debug.Print N1 >= N2    'True
    Debug.Print N1 = N2     'False
    Debug.Print N1 <> N2    'True
    Debug.Print S1$ < S2$   'True
    Debug.Print S1$ <= S2$  'True
    Debug.Print S1$ > S2$   'False
    Debug.Print S1$ >= S2$  'False
    Debug.Print S1$ = S2$   'False
    Debug.Print S1$ <> S2$  'True
    Debug.Print N1 And N2   ' 2
    Debug.Print N1 Or N2    ' 11
    Debug.Print N1 Xor N2   ' 9
    Debug.Print N1 Eqv N2   ' -10
    Debug.Print N1 Imp N2   ' -9
End Sub
```

Data Types

Any, Boolean, Byte, Currency, Date, Double, Integer, Long, Object, Single, String, String*n, Variant, user type.

Type	Description
Any	Any variable expression (**Declare** only).
Boolean	A *True* or *False* value.
Byte	An 8 bit unsigned integer value.
Cdec	Convert a number or string value to a 96 bit scaled real.
Currency	A 64 bit fixed point real. (A twos complement binary value scaled by 10000.)

Date	A 64 bit real value. The whole part represents the date, while the fractional part is the time of day. (December 30, 1899 = 0.) Use #date# as a literal date value in a macro.
Double	A 64 bit real value.
Integer	A 16 bit integer value.
Long	A 32 bit integer value.
Object	An object reference value. (see Objects)
PortInt	A portable integer value. For Win16: A 16 bit integer value. For Win32: A 32 bit integer value.
Single	A 32 bit real value.
String	An arbitrary length string value.
*String*n*	A fixed length (n) string value.
UserDialog	A *usertype* defined by **Begin Dialog** UserDialog.
Variant	An empty, numeric, currency, date, string, object, error code, null or array value.

Keywords

Empty, False, Nothing, Null, True. Win16, Win32.

Word	Description
Empty	A *variantvar* that does not have any value.
False	A *condexpr* is false when its value is zero. A function that returns False returns the value 0.
Nothing	An *objexpr* that does not refer to any object.
Null	An *variant expression* that is null. A null value propagates through an expression causing the entire expression to be Null. Attempting to use a Null value as a string or numeric argument causes a run-time error. A Null value prints as #NULL#.

Example

```
Sub Main
   X = Null
   Debug.Print X = Null '(even this expression is Null)
   Debug.Print IsNull(X) '(use IsNull to test for a _
   Null value)
End Sub
```

Example Output

```
Null
True
```

True	A *conditional expression* is true when its value is non-zero. A function that returns *True* returns the value -1.
Win16	**True** if running in 16 bits. **False** if running in 32 bits.
Win32	**True** if running in 32 bits. **False** if running in 16 bits.

Language Commands

Abs — Function

Syntax **Abs(***num***)**

Parameters	Name	Description
	num	Return the absolute value of this number value.

Description Return the absolute value.

Example

```
Sub Main
   Debug.Print Abs(9)
   Debug.Print Abs(0)
   Debug.Print Abs(-9)
End Sub
```

Example Output

```
 9
 0
 9
```

AppActivate — Instruction

Syntax **AppActivate** *title$*

-or-

AppActivate *TaskID*

Parameters	Name	Description
	title$	The name shown in the title bar of the window.
	TaskID	This numeric value is the task identifier.

Description Form 1: Activate the application top-level window titled Title$. If no window by that title exists then the first window with a title that starts with Title$ is activated. If no window matches then an error occurs.

Form 2: Activate the application top-level window for task TaskID. If no window for that task exists then an error occurs.

See Also SendKeys, Shell( ).

Example

```
Sub Main
   'Make ProgMan the active application
   AppActivate "Program Manager"
End Sub
```

Array — Function

Syntax `Array([expr[, ...]])`

Description Return a variant value array containing *exprs*.

Example

```
Sub Main
   X = Array(0,1,4,9)
   Debug.Print X(2)
End Sub
```

Example Output

```
4
```

Asc — Function

Syntax `Asc(string$)`

Parameters

Name	Description
string$	Return the ASCII value of the first char in this string value.

Description Return the ASCII value.

Note: A similar function, AscB, returns the first byte in S$. Another similar function, AscW, returns the Unicode number.

See Also Chr$( ).

Example

```
Sub Main
   Debug.Print Asc("A")
End Sub
```

Example Output

```
65
```

Atn — Function

Syntax `Atn(num)`

Parameters

Name	Description
num	Return the arc tangent of this number value. This is the number of radians. There are 2*Pi radians in a full circle.

Description Return the arc tangent.

Example

```
Sub Main
   Debug.Print Atn(1)*4
End Sub
```

Example Output

```
 3.14159265358979
```

Attribute

Definintion/Statement

Syntax **Attribute name =** *value*

Description All attribute definitions and statements are ignored except for:

- Public varname As Type
  Attribute varname.VB_VarUserMemId = 0

Declares Public varname as the default property for a class module or object module.

- Property [Get|Let|Set] propname (...)
  Attribute propname.VB_UserMemId = 0

  ...

End Property

Declares Property propname as the default property for a class module or object module.

Beep

Instruction

Syntax **Beep**

Description Sound the bell.

Example

```
Sub Main
   Beep          'Beep the bell.
End Sub
```

Begin Dialog

Definition

Syntax

```
Begin Dialog UserDialog [x, y,] dx, dy[, title$][,
.dialogfunc]
     User Dialog Item
     [User Dialog Item]...
End Dialog
```

Parameters

Name	Description
x	This number value is the distance from the left edge of the screen to the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font. If this is omitted then the dialog will be centered.
y	This number value is the distance from the top edge of the screen to the top edge of the dialog box. It is measured in 1/12ths of the

	average character width for the dialog's font. If this is omitted then the dialog will be centered.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
title$	This string value is the title of the user dialog. If this is omitted then there is no title.
dialogfunc	This is the function name that implements the **DialogFunc** for this UserDialog. If this is omitted then the *UserDialog* doesn't have a dialogfunc.
User Dialog Item	One of: CancelButton, CheckBox, ComboBox, DropListBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, PushButton, Text, TextBox.

Description Define a *UserDialog* type to be used later in a **Dim** As *UserDialog* statement.

See Also `Dim As UserDialog.`

Example

```
Sub Main
   Begin Dialog UserDialog 200,120
      Text 10,10,180,15,"Please push the OK button."
      OKButton 80,90,40,20
   End Dialog
   Dim dlg As UserDialog
   Dialog dlg  show dialog (Wait for OK)
End Sub
```

Call — Instruction

Syntax

```
Call name[(arglist)]
-or-
name[arglist]
```

Description Evaluate the *arglist* and call subroutine (or function) *name* with those values. Sub (or function) *name* must be previously defined by either a **Sub** (or **Function)** definition. If *name* is a function then the result is discarded. If Call is omitted then *name* must be a subroutine and the *arglist* is not enclosed in parens.

See Also **Declare, Sub.**

Example

```
Sub Show(Title$,Value)
   Debug.Print Title$;" =";Value
End Sub
Sub Main
```

```
    Call Show("2000/9",2000/9)
    Show "1",1<2        'True
End Sub
```

Example Output

```
222.2222222222
True
```

CallByName — Instruction

Syntax `CallByName(Obj,ProcName,CallType,[expr[, ...]])`

Description Call an Obj's method/property, ProcName, by name. Pass the exprs to the method/property.

Parameters

Name	Description
Obj	Call the method/property for this object reference.
ProcName	This string value is the name of the method/property to be called.
CallType	Type of method/property call. See table below.
expr	These expressions are passed to the obj's method/property.

CallType	Value	Effect
vbMethod	1	Call or evaluate the method.
vbGet	2	Evaluate the property's value.
vbLet	4	Assign the property's value.
vbSet	8	Set the property's reference.

Example

```
Sub Main
    On Error Resume Next
    CallByName Err, "Raise", vbMethod, 1
    Debug.Print CallByName(Err, "Number", vbGet) ' 1
End Sub
```

CallersLine — Function

Syntax `CallersLine[(Depth)]`

Description Return the caller's line as a text string.

The text format is: "[macroname|subname#linenum] linetext".

Parameter	Description
Depth	This integer value indicates how deep into the stack to get the caller's line. If Depth = 0 then return the current line. If Depth = 1 then return the calling subroutine's current line, etc. If Depth is greater than the call stack then a null string is returned. If this value is omitted then the depth is 1.

Example

```
Sub Main
    A
End Sub
Sub A
    Debug.Print CallersLine ' "[(untitled 1)|Main#  2] A"
End Sub
```

CancelButton Dialog Item

Definition

Syntax **CancelButton** *x, y, dx, dy*[, *.field*]

Parameters

Name	Description
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
field	This identifier is the name of the field. The dialogfunc receives this name as string. If this is omitted then the field name is Cancel.

Description Define a cancel button item. Pressing the Cancel button from a **Dialog** instruction causes a run-time error. (**Dialog**( ) function call returns 0.)

See Also Begin Dialog, Dim As UserDialog.

Example

```
Sub Main
  Begin Dialog UserDialog 200,120
     Text 10,10,180,30,"Please push the Cancel button"
     OKButton 40,90,40,20
     CancelButton 110,90,60,20
  End Dialog
  Dim dlg As UserDialog
  Dialog dlg  show dialog (wait for cancel)
  Debug.Print "Cancel was not pressed"
```

```
End Sub
```

CBool — Function

Syntax **CBool(***num|$***)**

Parameters

Name	Description
num	Any number.
$	The string must be either a number in quotes, or True or False in quotes (not case sensitive).

Description Convert to a boolean value. Zero converts to *False*, while all other values convert to *True*.

Example

```
Sub Main
   Debug.Print CBool(-1)
   Debug.Print CBool(0)
   Debug.Print CBool(1)
End Sub
```

Example Output

```
True
False
True
```

CByte — Function

Syntax `Byte(num|$)`

Parameters	Name	Description
	num|$	Convert a number or string value to a byte value.

Description Convert to a byte value.

Example

```
Sub Main
   Debug.Print CByte(1.6)
End Sub
```

Example Output

```
 2
```

CCur — Function

Syntax `CCur(num|$)`

Parameters	Name	Description
	num|$	Convert a number or string value to a currency value.

Description Convert to a *currency* value.

Example

```
Sub Main
   Debug.Print CCur(1E6)
End Sub
```

Example Output

```
 1000000
```

CDate — Function

Syntax `CDate(num|$)`
-or-
`CVDate(num|$)`

Parameters	Name	Description
	num|$	Convert a number or string value to a date value.

Description Convert to a *date* value.

Example

```
Sub Main
   Debug.Print CDate(2)
End Sub
```

Example Output

```
1/1/00
```

CDbl — Function

Syntax **CDbl(***num|$***)**

Parameters

Name	Description
num|$	Convert a number or string value to a double precision real.

Description Convert to a *double* precision real.

Example

```
Sub Main
   Debug.Print CDbl("1E6")
End Sub
```

Example Output

```
 1000000
```

ChDir — Instruction

Syntax **ChDir** *name$*

Parameters

Name	Description
name$	This string value is the path and name of the directory.

Description Change the current directory to *Name$*.

See Also ChDrive, CurDir$( ).

Example

```
Sub Main
   ChDir "C:\"
   Debug.Print CurDir$()
End Sub
```

Example Output

```
C:\
```

ChDrive — Instruction

Syntax **ChDrive** *drive$*

Parameters

Name	Description
drive$	This string value is the drive letter.

Description Change the current drive to *dfrive$*.

See Also ChDir, CurDir$( ).

Example

```
Sub Main
    ChDrive "B"
    Debug.Print CurDir$()
```

Example Output

```
B:\
```

CheckBox — Dialog Item Definition

Syntax `CheckBox x, y, dx, dy, title$, .field[, Options]`

Parameters

Name	Description
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
field	The value of the check box is accessed via this field. Checked is 1, and unchecked is 0.
Options	If this numeric value is zero or omitted then an checked/unchecked checkbox is created. If it is one then a checked/unchecked/grayed checkbox is created. If it is two then a checked/unchecked/grayed checkbox is created and the user can cycle through all three states.

Description Define a checkbox item.

See Also `Begin Dialog, Dim As UserDialog.`

Example

```
Sub Main
   Begin Dialog UserDialog 200,120
      Text 10,10,180,15,"Please push the OK button."
      CheckBox 10,25,180,15,"&Checkbox",.Check
      OKButton 80,90,40,20
   End Dialog
   Dim dlg As UserDialog
   dlg.Check = 1
   Dialog dlg             'Show dialog (wait for OK)
   Debug.Print dlg.Check
End Sub
```

Example Output

```
0
```

or

```
1
```

Choose — Function

Syntax `Choose(index, expr[, ...])`

Parameters

Name	Description
index	The numeric value indicates which *expr* to return. If this value is less than one or greater than the number of *exprs* then *Null* is returned.
expr	All expressions are evaluated.

Description Return the value of the *expr* indicated by *Index*.

See Also `If, Select Case, IIf( ).`

Example

```
Sub Main
   Debug.Print Choose(2,"Hi","there")
End Sub
```

Example Output

```
there
```

Chr$ Function

Syntax **Chr**[$]**(***num***)**

Parameters

Name	Description
num	Return one char string for this ASCII number value.

Description Return a one char string for the ASCII value.

Note: A similar function, ChrB, returns a single byte ASCII string. Another similar function, ChrW, returns a single char Unicode string.

See Also `Asc( ).`

Example

```
Sub Main
   Debug.Print Chr$(48)
End Sub
```

Example Output

```
0
```

CInt Function

Syntax **CInt(***num|$***)**

Parameters

Name	Description
num|$	Convert a number or string value to a 16 bit integer.

Description Convert to a 16 bit *integer*. If *num|$* is too big (or too small) to fit then an overflow error occurs.

Example

```
Sub Main
   Debug.Print CInt(1.6)
End Sub
```

Example Output 2

Class Module

Description: (The Class module feature is not implemented in version 1.5 of AP Basic)

A class module implements an OLE Automation object.

- Has a set of Public properties, functions and subroutines accessible from other macros and modules.
- These public symbols are accessed via an object variable.
- Public Consts, Types, arrays, fixed length strings are not allowed.
- A class module is similar to a object module except that no instance is automatically created.
- To create an instance use:

```
Dim Obj As classname
Set Obj = New classname
```

See Also Code Module, Object Module, Uses.

Example

```
'A.WWB
'#Uses "File.CLS"
Sub Main
    Dim File As New File
    File.Attach "C:\AUTOEXEC.BAT"
    Debug.Print File.ReadLine
End Sub

'File.CLS
'File|New Module|Class Module
'Edit|Properties|Name=File
Option Explicit
Dim FN As Integer
Public Sub Attach(FileName As String)
    FN = FreeFile
    Open FileName For Input As #FN
End Sub
Public Sub Detach()
    If FN <> 0 Then Close #FN
    FN = 0
End Sub
```

```
Public Function ReadLine() As String
    Line Input #FN,ReadLine
End Function

Private Sub Class_Initialize()
    Debug.Print "Class_Initialize"
End Sub

Private Sub Class_Terminate()
    Debug.Print "Class_Terminate"
    Detach
End Sub
```

Class_Initialize — Sub

Syntax

```
Private Sub Class_Initialize()
    ...
End Sub
```

Description Class module initialization subroutine. Each time a new instance is created for a class module the Class_Initialize sub is called. If Class_Initialize is not defined then no special initialization occurs.

See Also Code Module, Class_Terminate.

Class_Terminate — Sub

Syntax

```
Private Sub Class_Terminate()
    ...
End Sub
```

Description Class module termination subroutine. Each time an instance is destroyed for a class module the Class_Terminate sub is called. If Class_Terminate is not defined then no special termination occurs.

See Also Code Module, Class_Initialize.

Clipboard — Instruction/Function

Syntax

```
Clipboard text$
-or-
Clipboard[$][( )]
```

Parameters

Name	Description
text$	Put this string value into the clipboard.

Description

Form 1: Set the clipboard to Text$. This is like the Edit|Copy menu command.

Form 2: Return the text in the clipboard.

Example

```
Sub Main
   Debug.Print Clipboard$()
   Clipboard "Hello"
   Debug.Print Clipboard$()
End Sub
```

Example Output

```
Hello
```

CLng — Function

Syntax

```
CLng(num|$)
```

Parameters

Name	Description
num|$	Convert a number or string value to a 32 bit integer.

Description

Convert to a 32 bit *long* integer. If *num|$* is too big (or too small) to fit then an overflow error occurs.

Example

```
Sub Main
   Debug.Print CLng(1.6)
End Sub
```

Example Output

```
 2
```

Close — Instruction

Syntax

```
Close [[#]streamnum][, ...]
```

Parameters

Name	Description
streamnum	Streams 1, 2, 3 and 4 are available in each macro. If this is omitted then all open streams for the current macro are closed.

Description

Close *streamnums*.

See Also

Open, Reset.

Example

```
Sub Main
   'Read the first line of XXX and print it.
   Open "C:\APWIN\SAMPLES\SYSTEM1.APB" For Input As #1
   Line Input #1,L$
   Debug.Print L$
   Close #1
End Sub
```

Code

Module

Description (The Code module feature is not implemented in version 1.5 of AP Basic). A Code module implements a code library.

- Has a set of Public properties, functions and subroutines accessible from other macros and modules.
- The public symbols are accessed directly.

See Also `Class Module, Object Module, Uses.`

ComboBox

Dialog Item Definition

Syntax **ComboBox** *x, y, dx, dy, strarray$*( ), *.field$*

Parameters

Name	Description
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
strarray$( )	This one-dimensional array of strings establishes the list of choices. All the non-null elements of the array are used.
field$	The value of the combo box is accessed via this field. This is the text in the edit box .

Description Define a combobox item. Combo boxes combine the functionality of an edit box and a list box.

See Also `Begin Dialog, Dim As` *`UserDialog.`*

Example

```
Sub Main
```

```
Dim combos$(3)
   combos$(0) = "Combo 0"
   combos$(1) = "Combo 1"
   combos$(2) = "Combo 2"
   combos$(3) = "Combo 3"
   Begin Dialog UserDialog 200,120
      Text 10,10,180,15,"Please push the OK button"
      ComboBox 10,25,180,60,combos$(),.combo$
      OKButton 80,90,40,20
   End Dialog
   Dim dlg As UserDialog
   dlg.combo$ = none
   Dialog dlg          'show Dialog (Wait For ok)
   Debug.Print dlg.combo$
End Sub
```

Example Output

```
Combo 0
```

or

```
Combo 1
```

or

```
Combo 2
```

or

```
Combo 3
```

Command$ — Function

Syntax **Command**[$]

Description Contains the value of the **MacroRun** parameters.

See Also MacroRun

Example

```
Sub Main              'Macro 1 Calling Macro.
   MacroRun "MACRO2.APB 1,2,3"
End Sub
Sub Main              'Macro 2 "MACRO2.APB"
   Debug.Print "Command line parameter is: ";
   Debug.Print Command$;
End Sub
```

Example Output

```
Command line parameter is: 1,2,3
```

Const — Definition

Syntax `[Private|Public] Const name[type] [As type] = expr[, ...]`

Description Define *name* as the value of *expr*. The *expr* may refer to other constants or built-in functions. If the type of the constants is not specified, the type of *expr* is used. Constants defined outside a **Sub**, **Function** or **Property** block are available in the entire macro.

Private is assumed if neither *Private* or *Public* is specified.

Note: Const statements in a **Sub**, **Function** or **Property** block may not use *Private* or *Public*.

Example

```
Sub Main
Const Pi = 4*Atn(1), e = Exp(1)
   Debug.Print Pi
   Debug.Print e
End Sub
```

Example Output

```
 3.14159265358979
 2.71828182845905
```

Cos — Function

Syntax `Cos(num)`

Parameters

Name	Description
num	Return the cosine of this number value. This is the number of radians. There are 2*Pi radians in a full circle.

Description Return the cosine.

Example

```
Sub Main
   Debug.Print Cos(1)
End Sub
```

Example Output

```
 0.54030230586814
```

CreateObject — Function

Syntax `CreateObject(class$)`

Parameters

Name	Description
class$	This string value is the applications registered class name. If this application is not currently active it will be started.

Description Create a new object of type *Class$*. Use **Set** to assign the returned object to an object variable.

See Also `Objects`

Example

```
Sub Main
   Dim Excel As Object
   Set Excel = CreateObject("Excel.Application")
   With Excel
      Excel.Visible = True
      Excel.Quit
   End With
   Set Excel = Nothing
End Sub
```

CSng — Function

Syntax **CSng(***num|$***)**

Parameters	Name	Description
	num|$	Convert a number or string value to a single precision real.

Description Convert to a *single* precision real. If *num|$* is too big (or too small) to fit then an overflow error occurs.

Example

```
Sub Main
   Debug.Print CSng(Sqr(2))
End Sub
```

Example Output

```
 1.414214
```

CStr — Function

Syntax **CStr(***num|$***)**

Parameters	Name	Description
	num|$	Convert a number or string value to a string value.

Description Convert to a string.

Example

```
Sub Main
   Debug.Print CStr(Sqr(2))
End Sub
```

Example Output

```
1.4142135623731
```

CurDir$ Function

Syntax **CurDir**[$]**(**[*drive$*]**)**

Parameters

Name	Description
drive$	This string value is the drive letter. If this is omitted or null then return the current directory for the current drive.

Description Return the current directory for *Drive$*.

See Also ChDir, ChDrive.

Example

```
Sub Main
   Debug.Print CurDir$()
End Sub
```

Example Output

```
C:\
```

CVar Function

Syntax **CVar(***num|$***)**

Parameters

Name	Description
num|$	Convert a number or string value (or object reference) to a variant value.

Description Convert to a *variant value*.

Example

```
Sub Main
   Debug.Print CVar(Sqr(2))
End Sub
```

Example Output

```
 1.4142135623731
```

CVErr — Function

Syntax **CVErr(***num|$***)**

Parameters

Name	Description
num|$	Convert a number or string value to an error code.

Description Convert to a *variant* that contains an error code. An error code cant be used in expressions.

See Also IsError.

Example

```
Sub Main
   Debug.Print CVErr(1)
End Sub
```

Example Output

```
Error 1
```

Date — Function

Syntax **Date**[$]

Description Return today's *date* as a date value.

See Also Now, Time, Timer.

Example

```
Sub Main
   Debug.Print Date
End Sub
```

Example Output

```
2/8/96
```

DateAdd Function

Syntax **DateAdd(***inteval, number, dateexpr***)**

Description Return a date value a number of intervals from another date.

Parameter	Description
interval	This string value indicates which kind of interval to add.
number	Add this many intervals. Use a negative value to get an earlier date.
dateexpr	Calculate the new date relative to this date value. If this value is Null then Null is returned.

Interval	Description
yyyy	Year
q	Quarter
m	Month
d	Day
w	Weekday
ww	Week
h	Hour
m	Minute
s	Second

See Also DateDiff, DatePart.

Example

```
Sub Main
    Debug.Print DateAdd("yyyy",1,#1/1/2000#) '1/1/2001
End Sub
```

DateDiff Function

Syntax

DateDiff(*inteval, dateexpr1, dateexpr2***)**

Description Return the number of intervals between two dates.

Parameter	Description
interval	This string value indicates which kind of interval to subtract.
dateexpr1	Calculate the from this date value to dateexpr2. If this value is Null then Null is returned.
dateexpr2	Calculate the from dateexpr1 to this date value. If this value is Null then Null is returned.

Interval	Description
yyyy	Year
q	Quarter
m	Month
d	Day
w	Weekday
ww	Week
h	Hour
m	Minute
s	Second

See Also DateAdd, DatePart.

Example

```
Sub Main
    Debug.Print DateDiff("yyyy",#1/1/1990#,#1/1/2000#) '
10
End Sub
```

DatePart Function

Syntax **DatePart(***inteval, dateexpr***)**

Description Return the number from the date corresponding to the interval.

Parameter	Description
interval	This string value indicates which kind of interval to extract.
dateexpr	Get the interval from this date value. If this value is Null then Null is returned.

Interval	Description (return value range)
yyyy	Year (100-9999)
q	Quarter (1-4)
m	Month (1-12)

d	Day (1-366)
w	Weekday (1-7)
ww	Week (1-53)
h	Hour (0-23)
m	Minute (0-59)
s	Second (0-59)

See Also DateAdd, DateDiff.

Example

```
Sub Main
    Debug.Print DatePart("yyyy",#1/1/2000#) ' 2000
End Sub
```

DateSerial — Function

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

Parameters

Name	Description
year	This numeric value is the year (0 to 9999). (0 to 99 are interpreted as 1900 to 1999.)
month	This numeric value is the month (1 to 12).
day	This numeric value is the day (1 to 31).

Description Return a *date* value.

See Also DateValue, TimeSerial, TimeValue.

Example

```
Sub Main
   Debug.Print DateSerial(1996,2,8)
End Sub
```

Example Output

```
2/8/9
```

DateValue — Function

Syntax **DateValue(***date$***)**

Parameters

Name	Description
date$	Convert this string value to the day part of date it represents.

Description Return the day part of the date encoded as a string.

See Also DateSerial, TimeSerial, TimeValue.

Example

```
Sub Main
   Debug.Print DateValue("2/8/1996 12:00:01 AM")
End Sub
```

Example Output

```
2/8/96
```

Day

Function

Syntax **Day(***dateexpr***)**

Parameters

Name	Description
dateexpr	Return the day of the month for this date value.

Description Return the day of the month (1 to 31).

See Also Date( ), Month( ), Weekday( ), Year( ).

Example

```
Sub Main
   Debug.Print Day(#1/1/1900#)
End Sub
```

Example Output

```
 1
```

dBToPowerRatio

Function

Syntax **dBToPowerRatio(***num***)**

Parameters

Name	Description
num	dB number

Description Return the power ratio of *num* to 1.

Example

```
Sub Main
   Debug.Print Format(dBToPowerRatio(-3), "#.0000")
End Sub
```

Example Output

```
.5012
```

Equation

```
PowerRatio = Exp10(num / 10)
```

dBToVoltageRatio

Function

Syntax **dBToVoltageRatio(***num***)**

Parameters

Name	Description
num	dB number

Description Return the voltage ratio of *num* to 1.

Example

```
Sub Main
   Debug.Print Format(dBToVoltageRatio(-6), "#.0000")
End Sub
```

Example Output

```
.5012
```

Equation

```
VoltageRatio = Exp10(num/20)
```

DDEExecute — Instruction

Syntax **DDEExecute** *channum*, command$[, *timeout*]

Parameters

Name	Description
channum	This is the channel number returned by the **DDEInitiate** function. Up to 10 channels may be used at one time.
command$	Send this command value to the server application. The interpretation of this value is defined by the server application.
timeout	The command will generate an error if the number of seconds specified by the timeout is exceeded before the command has completed. The default is five seconds.

Description Send the DDE Execute *Command$* string via DDE *Channum.*

Example

```
Sub Main
   ChanNum = DDEInitiate(PROGMAN,"PROGMAN")
   DDEExecute ChanNum,"[CreateGroup(XXX)]"
   DDETerminate ChanNum
End Sub
```

DDEInitiate — Function

Syntax **DDEInitiate(***app$, topic$***)**

Parameters

Name	Description
app$	Locate this server application.
topic$	This is the server applications topic. The interpretation of this value is defined by the server application.

Description Initiate a DDE conversation with *App$* using *Topic$.* If the conversation is successfully started then the return value is a channel number that can be used with other DDE instructions and functions.

Example

```
Sub Main
   ChanNum = DDEInitiate (PROGMAN, PROGMAN)
   DDEExecute ChanNum,"[CreateGroup(XXX)]"
   DDETerminate ChanNum
End Sub
```

DDEPoke — Instruction

Syntax

```
DDEPoke channum, item$, data$[, timeout]
```

Parameters

Name	Description
channum	This is the channel number returned by the **DDEInitiate** function. Up to 10 channels may be used at one time.
item$	This is the server applications item. The interpretation of this value is defined by the server application.
data$	Send this data value to the server application. The interpretation of this value is defined by the server application.
timeout	The command will generate an error if the number of seconds specified by the timeout is exceeded before the command has completed. The default is five seconds.

Description

Poke *Data$* to the *Item$* via DDE *Channum*.

Example

```
Sub Main
   ChanNum = DDEInitiate(PROGMAN,"PROGMAN")
   DDEPoke ChanNum,"Group","XXX"
progman doesn't support poke
   DDETerminate ChanNum
End Sub
```

DDERequest$ — Function

Syntax

```
DDERequest[$](channum, item$[, timeout])
```

Parameters

Name	Description
channum	This is the channel number returned by the **DDEInitiate** function. Up to 10 channels may be used at one time.
item$	This is the server applications item. The interpretation of this value is defined by the server application.
timeout	The command will generate an error if the number of seconds specified by the timeout is exceeded before the command has completed. The default is five seconds.

Description

Request information for *Item$*. If the request is not satisfied then the return value will be a null string.

Example

```
Sub Main
   ChanNum = DDEInitiate(PROGMAN,"PROGMAN")
   Debug.Print DDERequest$(ChanNum,"Groups")
   DDETerminate ChanNum
End Sub
```

DDETerminate — Instruction

Syntax **DDETerminate** *channum*

Parameters

Name	Description
channum	This is the channel number returned by the **DDEInitiate** function. Up to 10 channels may be used at one time.

Description Terminate DDE *Channum*.

Example

```
Sub Main
   ChanNum = DDEInitiate(PROGMAN,"PROGMAN")
   DDEExecute ChanNum,"[CreateGroup(XXX)]"
   DDETerminate ChanNum
End Sub
```

DDETerminateAll — Instruction

Syntax **DDETerminateAll**

Description Terminate all open DDE channels.

Example

```
Sub Main
   ChanNum = DDEInitiate(PROGMAN,"PROGMAN")
   DDEExecute ChanNum,"[CreateGroup(XXX)]"
   DDETerminateAll
End Sub
```

Debug — Object

Syntax

```
Debug.Print [expr[; ...][;]]
```

Description

Print the *expr*(s) to the output window. Use ; to separate expressions. A *num* is automatically converted to a string before printing (just like **Str$**( )). If the instruction does not end with a ; then a newline is printed at the end.

Example

```
Sub Main
   X = 4
   Debug.Print "X/2 ="; X/2
   Debug.Print "Start...";   'Dont Print
a newline
   Debug.Print "Finish"      'Print a newline"
End Sub
```

Example Output

```
X/2 = 2
Start...Finish
```

Declare — Definition

Syntax

```
[Private|Public] Declare Sub name Lib dllname _
[Alias modulename] [([param[, ...]])]
-or-
[Private|Public] Declare Function name[type] Lib _
dllname [Alias modulename] [([param[, ...]])] As _
type]
```

Parameters

Name	Description
name	This is the name of the subroutine/function being defined.
dll name	This is the DLL file where the modules code is.
module name	This is the name of the module in the DLL file. If this is #number then it is the ordinal number of the module. If it is omitted then name is the module name.
params	A list of zero or more params that are used by the DLL subroutine or function. (Note : A ByVal strings value may be modified by the DLL.)

Description

Interface to a DLL defined subroutine or function. The values of the calling *arglist* are assigned to the *params*.

Public is assumed if neither *Private* or *Public* is specified.

WARNING! Be very careful when declaring DLL subroutines or functions. If you make a mistake and declare the paremeters or result incorrectly then

Windows might halt. Save any open documents before testing new DLL declarations.

See Also `Function, Sub, Call.`

Example

```
Declare Function GetActiveWindow& Lib "user32" ()
Declare Function GetWindowTextLength% Lib "user32" _ (ByVal hwnd&)
Declare Sub GetWindowText Lib "user32"
(ByVal hwn&%, _ ByVal lpsz$, ByVal cbMax&)
Function ActiveWindowTitle$()
   ActiveWindow = GetActiveWindow()
   TitleLen =   GetWindowTextLength(ActiveWindow)
   Title$ = Space$(TitleLen)
   GetWindowText   ActiveWindow,Title$,TitleLen+1
   ActiveWindowTitle$ = Title$
End Function
Sub Main
   Debug.Print ActiveWindowTitle$()
End Sub
```

Def

Definition

Syntax **Def**{Bool|Cur|Date|Dbl|Int|Lng|Obj|Sng|Str|Var} *letterrange*[, ...]

Parameters

Name	Description
letterrange	letter, or letter-letter: A letter is one of A to Z. When letter-letter is used, the first letter must be alphabetically before the second letter. Variable names that begin with a letter in this range default to declared type. If a variable name begins with a letter not specific in any letterrange then the variable is a Variant. The letterranges are not allowed to overlap.

Description Define untyped variables as:

DefBool - *Boolean*

DefByte - *Byte*

DefCur - *Currency*

DefDate - *Date*

DefDbl - *Double*

DefInt - *Integer*

DefLng - *Long*

DefObj - *Object*

DefSng - *Single*

DefStr - *String*

DefVar - *Variant*

See Also `Option Explicit.`

Example

```
DefInt A,C-W,Y     'Integers
DefBool B             'Boolean
DefStr X              'String
                      'All others(Z) are Variant.

Sub Main
   B = 1              'B Is an Boolean.
   Debug.Print B
   X = "A"            'X Is a String.
   Debug.Print X
   Z = 1              'Z Is a Variant (anything).
   Debug.Print Z
   Z = "Z"
   Debug.Print Z
End Sub
```

Example Output

```
 1
A
 1
Z
```

DeleteSetting — Instruction

Syntax

```
DeleteSetting AppName$, Section$[, Key$]
```

Description

Delete the settings for Key in Section in project AppName. Win16 and Win32s store settings in an .ini file named AppName. Win32 stores settings in the registration database.

Parameter	Description
AppName$	This string value is the name of the project that has this Section and Key.
Section$	This string value is the name of the section of the project settings.
Key$	This string value is the name of the key in the section of the project settings. If this is omitted then delete the entire section.

Example

```
Sub Main
    SaveSetting "MyApp","Font","Size",10
    DeleteSetting "MyApp","Font","Size"
End Sub
```

Dialog — Instruction/Function

Syntax

```
Dialog dialogvar[, default]
-or-
Dialog(dialogvar[, default])
```

Parameters

Name	Description
dlgvar	This variable that holds the values of the fields in a dialog. Use .field to access individual fields in a dialog variable.
default	This numeric value indicates which button is the default button. (Pressing the Enter key on a non-button pushes the default button.) Use -2 to indicate that there is no default button. Other possible values are shown the result table below. If this value is omitted then the first **PushButton**, **OKButton** or **CancelButton** is the default button.

Result

Value	Description
-1	**OK button** was pressed.
0	**Cancel button** was pressed
n	Nth **push button** was pressed.

Description

Display the dialog associated with *dialogvar*. The initial values of the dialog fields are provided by dialogvar. If the **OK button** or any **push button** is pressed then the fields in dialog are copied to the *dialogvar*. The Dialog( )

function returns a value indicating which button was pressed. (See the result table below.)

See Also `Begin Dialog, Dim As UserDialog.`

Example

```
Sub Main
   Begin Dialog UserDialog 200,120
      Text 10,10,180,15,"Please push the OK button."
      OKButton 80,90,40,20
   End Dialog
   Dim dlg As UserDialog
   Dialog dlg          'Show Dialog (Wait For OK)
End Sub
```

DialogFunc

Prototype

Syntax

```
Function Dialogfunc(dlgitem$, action%, suppvalue%) _ As
Boolean
     Select Case Action%
     Case 1  Dialog box initialization
     ...
     Case 2  Value changing or button pressed
     ...
     Case 3  TextBox or ComboBox text changed
     ...
     Case 4  Focus changed
     ...
     Case 5  Idle
     ...
     End Select
End Function
```

Parameters

Name	Description
dlgitem	This string value is the name of the user dialog items field.
action	This numeric value indicates what action the dialog function is being asked to do.
suppvalue	This numeric value provides additional information for some actions.

Action	Description
1	Dialog box initialization. *DlgItem* is a null string. *SuppValue* is zero.
2	**CheckBox, DropListBox, ListBox** or **OptionGroup**: *DlgItem*s value has changed. *SuppValue* is the new value.

	CancelButton, OKButton or **PushButton**: *DlgItem*s button was pushed. *SuppValue* is meaningless. Set *dialogfunc* = **True** to prevent the dialog from closing.
3	**ComboBox** or **TextBox**: *DlgItem*s text changed and losing focus. *SuppValue* is the number of characters.
4	Item *DlgItem* is gaining focus. *SuppValue* is the item that is losing focus. (The first item is 0, second is 1, etc.)
5	Idle processing. *DlgItem* is a null string. *SuppValue* is zero. Set *dialogfunc* = **True** to continue receiving idle actions.

Description A dialogfunc implements the dynamic dialog capabilities.

See Also `Begin Dialog.`

Example

```
Sub Main
    Begin Dialog UserDialog 200,120,.DialogFunc
        Text 10,10,180,15,"Please push the OK button."
        TextBox 10,40,180,15,.Text
        OKButton 30,90,60,20
        PushButton 110,90,60,20,"&Hello"
    End Dialog
    Dim dlg As UserDialog
    Debug.Print Dialog(dlg)
End Sub
Function DialogFunc%(DlgItem$, Action%, SuppValue%)
    Debug.Print "Action =";Action%
    Select Case Action%
    Case 1
        Beep
    Case 2
        If DlgItem$ = "Hello" Then
            MsgBox "Hello"
            DialogFunc% = True 'do not exit the dialog
        End If
    Case 3
        Debug.Print DlgItem$;"=""";DlgText$(DlgItem$);"""
    Case 4
        Debug.Print "DlgFocus =""";DlgFocus();"""
    End Select
End Function
```

Dim

Definition

Syntax `Dim name[type][([Dim[,...]])][As type][, ...]`

Description Dimension var array(s) using the *dims* to establish the minimum and maximum index value for each dimension. If the *dims* are omitted then a scalar (single value) variable is defined. A dynamic array is declared using ( ) without any *dims.* It must be **ReDim**ensioned before it can be used.

See Also Begin Dialog, Dialog, Private, Public, ReDim, Static.

Example

```
Sub DoIt(Size)
   Dim C0,C1(),C2(2,3)
   ReDim C1(Size)      'Dynamic Array
   C0 = 1
   C1(0) = 2
   C2(0,0) = 3
   Debug.Print C0;C1(0);C2(0,0)
End Sub
Sub Main
   DoIt 1
End Sub
```

Example Output

```
 1 2 3
```

Dir$

Function

Syntax `Dir[$]([pattern$], [attribmask])`

Parameters

Name	Description
pattern$	This string value is the path and name of the file search pattern. If this is omitted then continue scanning with the previous pattern. Each macro has its own independent search. A path relative to the current directory can be used.
attribmask	This numeric value controls which files are found. A file with an attribute that matches will be found.

Description Scan a directory for the first file matching *Pattern$.*

See Also GetAttr( ).

Example

```
Sub Main
   F$ = Dir$("*.*")
   While F$ <> ""
      Debug.Print F$
      F$ = Dir$()
   Wend
```

```
End Sub
```

Example Output

```
SNR.APB
FRQ-RESP.AT1
READINGS.APB...
```

DlgControlId — Function

Syntax **DlgControlId(***dlgitem|$***)**

Parameters

Name	Description
dlgitem|$	If this is a numeric value then it is the dialog item number. The first item is 0, second is 1, etc. If this is a string value then it is the dialog items *field* name.

Description Return the *fields* window id.

This instruction/function must be called directly or indirectly from a *dialogfunc*.

Example

```
Sub Main
   Begin Dialog UserDialog 200,120, .DialogFunc
      Text 10,10,180,15,"Please push the OK button."
      TextBox 10,40,180,15,.Text
      OKButton 30,90,60,20
      PushButton 110,90,60,20,"&Hello"
   End Dialog
   Dim dlg As UserDialog
   Debug.Print Dialog(dlg)
End Sub
Function DialogFunc%(DlgItem$, Action%, SuppValue%)
   Debug.Print "Action =";Action%
   Select Case Action%
   Case 1              'Dialog box initialization
      Beep
   Case 2              'Value changing Or button pressed
      If DlgItem$ = Hello Then
         DialogFunc% = True 'Do Not Exit the Dialog
      End If
   Case 4              'Focused changed
      Debug.Print "DlgFocus = """;DlgFocus();""""
      Debug.Print "DlgControlId("; DlgItem$;") =";
      Debug.Print DlgControlId(DlgItem$)
   End Select
End Function
```

DlgCount — Function

Syntax `DlgCount()`

Description Return the number of dialog items in the dialog. This instruction/function must be called directly or indirectly from a *dialogfunc*.

Example

```
Sub Main
   Begin Dialog UserDialog 200,120,.DialogFunc
      Text 10,10,180,15,"Please push the OK button."
      TextBox 10,40,180,15,.Text
      OKButton 30,90,60,20
   End Dialog
   Dim dlg As UserDialog
   Dialog dlg
End Sub
Function DialogFunc%(DlgItem$, Action%, SuppValue%)
   Debug.Print "Action =";Action%
   Select Case Action%
   Case 1          'Dialog box initialization
      Beep
      Debug.Print "DlgCount =";DlgCount()3
   End Select
End Function
```

DlgEnable — Instruction/Function

Syntax

DlgEnable *dlgitem|$*[, *enable*]
-or-
DlgEnable(*dlgitem|$***)**

Parameters

Name	Description
dlgitem|$	If this is a numeric value then it is the dialog item number. The first item is 0, second is 1, etc. If this is a string value then it is the dialog item's field name. Note: Use -1 to enable or disable all the dialog items at once.
enable	It this numeric value is True then enable DlgItem|$. Otherwise, disable it. If this omitted then toggle it.

Description Instruction: Enable or disable *DlgItem|$*.

Function: Return True if *DlgItem|$* is enabled.

This instruction/function must be called directly or indirectly from a *dialogfunc*.

Example

```
Sub Main
   Begin Dialog UserDialog 200,120,.DialogFunc
      Text 10,10,180,15,"Please push the OK button to
exit."
      TextBox 10,40,180,15,.Text
      OKButton 30,90,60,20
      PushButton 110,90,60,20,"&Disable"
   End Dialog
   Dim dlg As UserDialog
   Debug.Print Dialog(dlg)
End Sub
Function DialogFunc%(DlgItem$, Action%, SuppValue%)
   Debug.Print "Action =";Action%
   Select Case Action%
   Case 1    'Dialog box initialization
      Beep
   Case 2    'Value changing Or button pressed
      Select Case DlgItem$
      Case "Disable"
         DlgText DlgItem$,"&Enable"
         DlgEnable Text,False
         DialogFunc% = True 'Do not exit the dialog.
      Case "Enable"
         DlgText DlgItem$,"&Disable"
         DlgEnable Text,True
         DialogFunc% = True 'Do not exit the dialog.
      End Select
   End Select
End Function
```

DlgEnd — Instruction

Syntax **DlgEnd ReturnCode**

Description Set the return code for the Dialog Function and close the user dialog.

This instruction/function must be called directly or indirectly from a dialogfunc.

Parameters

Parameter	Description
ReturnCode	Return this numeric value.

Example

```
Sub Main
    Begin Dialog UserDialog 210,120,.DialogFunc
```

```
            Text 10,10,190,15,"Please push the Close
button"
        OKButton 30,90,60,20
            CheckBox 120,90,60,20,"&Close",.CheckBox1
    End Dialog
    Dim dlg As UserDialog
    Debug.Print Dialog(dlg)
End Sub

Function DialogFunc%(DlgItem$, Action%, SuppValue%)
    Debug.Print "Action=";Action%
    Select Case Action%
    Case 1 ' Dialog box initialization
        Beep
    Case 2 ' Value changing or button pressed
        Select Case DlgItem$
        Case "CheckBox1"
            DlgEnd 1000
        End Select
    End Select
End Function
```

DlgFocus — Instruction/Function

Syntax

DlgFocus dlgitem|$
-or-
dlgfocus[$]**()**

Parameters

Name	Description
dlgitem|$	If this is a numeric value then it is the dialog item number. The first item is 0, second is 1, etc. If this is a string value then it is the dialog items *field* name.

Description

Instruction: Move the focus to this *DlgItem|$*.

Function: Return the *field* name which has the focus as a string.

This instruction/function must be called directly or indirectly from a *dialogfunc*.

Example

```
Sub Main
  Begin Dialog UserDialog 200,120,.DialogFunc
     Text 10,10,180,15,"Please push the OK button"
     TextBox 10,40,180,15,.Text
     OKButton 30,90,60,20
```

```
      PushButton 110,90,60,20,"&Hello"
   End Dialog
   Dim dlg As UserDialog
   Debug.Print Dialog(dlg)
End Sub
Function DialogFunc%(DlgItem$,Action%,SuppValue%)
   Debug.Print "Action =";Action%
   Select Case Action%
   Case 1    'Dialog box initialization
      Beep
   Case 2    'Value changing Or button pressed
      If DlgItem$ = "Hello" Then
         MsgBox "Hello Button Pressed"
         DialogFunc% = True 'Do Not Exit the Dialog
      End If
   Case 4    'Focus changed
      Debug.Print "DlgFocus =""";DlgFocus();"""
   End Select
End Function
```

Example Output

DlgListBoxArray — Instruction/Function

Syntax

DlgListBoxArray *dlgitem|$, strarray$()*
-or-
DlgListBoxArray(*dlgitem|$[, strarray$()]***)**

Parameters

Name	Description
dlgitem|$	If this is a numeric value then it is the dialog item number. The first item is 0, second is 1, etc. If this is a string value then it is the dialog items *field* name.
strarray$( )	Set the list entries of *DlgItem|$*. This one-dimensional array of strings establishes the list of choices. All the non-null elements of the array are used.

Description

Instruction: Set the list entries for *DlgItem|$*.

Function: Return the number entries in *DlgItem|$*s list.

This instruction/function must be called directly or indirectly from a *dialogfunc*. The *DlgItem|$* should refer to a **ComboBox**, **DropListBox** or **ListBox**.

Example

```
Dim lists$()
Sub Main
```

```
   ReDim lists$(0)
   lists$(0) = "List 0"
   Begin Dialog UserDialog 200,119,.DialogFunc
      Text 10,7,180,14,"Please push the OK button."
      ListBox 10,21,180,63,lists(),.list
      OKButton 30,91,40,21
      PushButton 110,91,60,21,"&Change"
   End Dialog
   Dim dlg As UserDialog
   dlg.list = 2
   Dialog dlg             'Show Dialog (Wait For ok)
   Debug.Print dlg.list
End Sub
Function DialogFunc%(DlgItem$, Action%, SuppValue%)
   Select Case Action%
   Case 2       'Value changing Or button pressed
      If DlgItem$ = "Change" Then
         Dim N As Integer
         N = UBound(lists$) + 1
         ReDim Preserve lists$(N)
         lists$(N) = "List " & N
         DlgListBoxArray "list",lists$()
         DialogFunc% = True 'Do Not Exit the Dialog
      End If
   End Select
End Function
```

DlgName Function

Syntax **DlgName**[$](*dlgitem*)

Parameters

Name	Description
dlgitem	This numeric value is the dialog item number. The first item is 0, second is 1, etc.

Description Return the *field* name of the *DlgItem* number. This instruction/function must be called directly or indirectly from a *dialogfunc*.

Example

```
Sub Main
   Begin Dialog UserDialog 200,120,.DialogFunc
      Text 10,10,180,15,"Please push the OK button.",.Text
      TextBox 10,40,180,15,.TextBox
      OKButton 30,90,60,20,.OKButton
   End Dialog
   Dim dlg As UserDialog
   Dialog dlg
End Sub
Function DialogFunc%(DlgItem$, Action%, SuppValue%)
   Debug.Print "Action =";Action%
   Select Case Action%
   Case 1       'Dialog box initialization.
      Beep
      For I = 0 To DlgCount()-1
         Debug.Print I ;" = ";DlgName(I)
      Next I
   End Select
End Function
```

Example Output

```
Action = 1
 0 = Text
 1 = TextBox
 2 = OKButton
Action = 4
Action = 5
Action = 4
Action = 2
```

DlgNumber Function

Syntax **DlgNumber**(*dlgitem$*)

Parameters

Name	Description
dlgitem$	This string value is the dialog items *field* name.

Description Return the number of the *DlgItem$*. The first item is 0, second is 1, etc. This instruction/function must be called directly or indirectly from a *dialogfunc*.

Example

```
Sub Main
   Begin Dialog UserDialog 200,120,.DialogFunc
      Text 10,10,180,15,"Please push the OK button."
      TextBox 10,40,180,15,.Text
      OKButton 30,90,60,20
   End Dialog
   Dim dlg As UserDialog
   Dialog dlg
End Sub
Function DialogFunc%(DlgItem$, Action%, SuppValue%)
   Debug.Print "Action =";Action%
   Select Case Action%
   Case 1          'Dialog box initialization
      Beep
   Case 4          'Focus changed
      Debug.Print DlgItem$;" =";DlgNumber(DlgItem$)
   End Select
End Function
```

Example Output

```
Action = 1
Action = 4
Text = 1
Action = 5
Action = 4
OK = 2
Action = 2
```

DlgSetPicture — Instruction

Syntax: **DlgSetPicture** *DlgItem|$, FileName, Type*

Description Instruction: Set the file name for DlgItem|$.

This instruction/function must be called directly or indirectly from a dialogfunc.

Parameters

Parameter	Description
DlgItem|$	If this is a numeric value then it is the dialog item number. The first item is 0, second is 1, etc. If this is a string value then it is the dialog item's field name.
FileName	Set the file name of DlgItem|$ to this string value.
Type	This numeric value indicates the type of bitmap used. See below.

Type	Effect
0	FileName is the name of the bitmap file. If the file does not exist then "(missing picture)" is displayed.
3	The clipboard's bitmap is dispalyed. Not supported.
+16	Instead of displaying "(missing picture)" a run-time error occurs.

Example

```
Sub Main
    Begin Dialog UserDialog 200,120,.DialogFunc
        Picture 10,10,180,75,"",0,.Picture
        OKButton 30,90,60,20
        PushButton 110,90,60,20,"&View"
    End Dialog
    Dim dlg As UserDialog
    Debug.Print Dialog(dlg)
End Sub

Function DialogFunc%(DlgItem$, Action%, SuppValue%)
    Debug.Print "Action=";Action%
    Select Case Action%
    Case 1 ' Dialog box initialization
        Beep
    Case 2 ' Value changing or button pressed
        Select Case DlgItem$
        Case "View"
            FileName = GetFilePath("Bitmap","BMP")
            DlgSetPicture "Picture",FileName,0
            DialogFunc% = True 'do not exit the dialog
        End Select
    End Select
End Function
```

DlgText

Instruction/Function

Syntax

DlgText *dlgitem|$, text*
-or-
DlgText[$]**(***dlgitem|$***)**

Parameters

Name	Description
dlgitem|$	If this is a numeric value then it is the dialog item number. The first item is 0, second is 1, etc. If this is a string value then it is the dialog items *field* name. Note: Use -1 to access the dialog's title.
text	Set the text of *DlgItem|$* to this string value.

Description

Instruction: Set the text for *DlgItem|$*.

Function: Return the text from *DlgItem|$*.

This instruction/function must be called directly or indirectly from a *dialogfunc*.

Example

```
Sub Main
    Begin Dialog UserDialog 200,120,.DialogFunc
        Text 10,10,180,15,"Please push the OK button."
        TextBox 10,40,180,15,.Text
        OKButton 30,90,60,20
        PushButton 110,90,60,20,"&Now"
    End Dialog
    Dim dlg As UserDialog
    Debug.Print Dialog(dlg)
End Sub

Function DialogFunc%(DlgItem$, Action%, SuppValue%)
    Debug.Print "Action =";Action%
    Select Case Action%
    Case 1
        Beep
    Case 2
        Select Case DlgItem$
        Case "Now"
            DlgText "Text",CStr(Now)
            DialogFunc% = True 'Do not exit the dialog
        End Select
    End Select
End Function
```

Example Output

```
Action = 1
Action = 4
Action = 5
```

```
Action = 4
Action = 2
-1
```

DlgType — Function

Syntax **DlgType**[$](*dlgitem|$*)

Parameters

Name	Description
dlgitem|$	If this is a numeric value then it is the dialog item number. The first item is 0, second is 1, etc. If this is a string value then it is the dialog items *field* name.

Description Return a string value indicating the type of the *DlgItem|$*. One of: **CancelButton**, **CheckBox**, **ComboBox**, **DropListBox**, **GroupBox**, **ListBox**, **OKButton**, **OptionButton**, **OptionGroup**, **PushButton**, **Text**, **TextBox**. This instruction/function must be called directly or indirectly from a *dialogfunc*.

Example

```
Sub Main
   Begin Dialog UserDialog 200,120,.DialogFunc
      Text 10,10,180,15,"Please push the OK button."
      TextBox 10,40,180,15,.Text
      OKButton 30,90,60,20
   End Dialog
   Dim dlg As UserDialog
   Dialog dlg
End Sub
Function DialogFunc%(DlgItem$, Action%, SuppValue%)
   Debug.Print Action=;Action%
   Select Case Action%
   Case 1  Dialog box initialization
      Beep
      For I = 0 To DlgCount()-1
         Debug.Print I;" ";DlgType(I)
      Next I
   End Select
End Function
```

Example Output

```
Action = 1
 0 Text
 1 TextBox
 2 OKButton
Action = 4
```

```
Action = 5
Action = 4
Action = 2
```

DlgValue — Instruction/Function

Syntax

DlgValue *dlgitem|$, value*
-or-
DlgValue(*dlgitem|$***)**

Parameters

Name	Description
dlgitem|$	If this is a numeric value then it is the dialog item number. The first item is 0, second is 1, etc. If this is a string value then it is the dialog items *field* name.
text	Set the text of *DlgItem|$* to this string value.

Description

Instruction: Set the numeric value *DlgItem|$*

Function: Return the numeric value for *DlgItem|$*.

This instruction/function must be called directly or indirectly from a *dialogfunc*. The *DlgItem|$* should refer to a **CheckBox**, **DropListBox**, **ListBox** or **OptionGroup**.

Example

```
Sub Main
   Begin Dialog UserDialog 150,147,.DialogFunc
      GroupBox 10,7,130,77,"Direction",.Field1
      PushButton 100,28,30,21,"&Up"
      PushButton 100,56,30,21,"&Dn"
      OptionGroup .Direction
         OptionButton 20,21,80,14,"&North",.North
         OptionButton 20,35,80,14,"&South",.South
         OptionButton 20,49,80,14,"&East",.East
         OptionButton 20,63,80,14,"&West",.West
      OKButton 10,91,130,21
      CancelButton 10,119,130,21
   End Dialog
   Dim dlg As UserDialog
   Dialog dlg
   MsgBox "Direction = " & dlg.Direction
End Sub
Function DialogFunc%(DlgItem$, Action%, SuppValue%)
   Select Case Action%
   Case 1          'Dialog box initialization.
```

```
      Beep
   Case 2          'Value changing Or button pressed.
      Select Case DlgItem$
      Case "Up"
         DlgValue "Direction",0
         DialogFunc% = True 'Do Not Exit the Dialog.
      Case "Dn"
         DlgValue "Direction",1
         DialogFunc% = True 'Do Not Exit the dialog.
      End Select
   End Select
End Function
```

DlgVisible

Instruction/Function

Syntax

DlgVisible dlgitem|$, *visible*
-or-
DlgVisible(*dlgitem|$***)**

Parameters

Name	Description
dlgitem|$	If this is a numeric value then it is the dialog item number. The first item is 0, second is 1, etc. If this is a string value then it is the dialog items *field* name.
enable	It this numeric value is non-zero then show *DlgItem|$*. Otherwise, hide it.

Description

Instruction: Show or hide *DlgItem|$*.

Function: Return *True* if *DlgItem|$* is visible.

This instruction/function must be called directly or indirectly from a *dialogfunc*.

Example

```
Sub Main
   Begin Dialog UserDialog 200,120,.DialogFunc
      Text 10,10,180,15,"Please push the OK button"
      TextBox 10,40,180,15,.Text
      OKButton 30,90,60,20
      PushButton 110,90,60,20,"&Hide"
   End Dialog
   Dim dlg As UserDialog
   Debug.Print Dialog(dlg)
End Sub

Function DialogFunc%(DlgItem$, Action%, SuppValue%)
   Debug.Print "Action =";Action%
   Select Case Action%
   Case 1          'Dialog box initialization
      Beep
   Case 2          'Value changing Or button pressed
      Select Case DlgItem$
      Case "Hide"
         DlgText DlgItem$,"&Show"
         DlgVisible "Text",False
         DialogFunc% = True 'Do Not Exit the Dialog
      Case "Show"
         DlgText DlgItem$,"&Hide"
         DlgVisible "Text",True
         DialogFunc% = True 'Do Not Exit the Dialog
```

```
        End Select
    End Select
End Function
```

Do Statement

Syntax

```
Do
    statements
Loop
-or-
Do {Until|While} condexpr
    statements
Loop
-or-
Do
    statements
Loop {Until|While} condexpr
```

Description

Form 1: Do *statements* forever. The loop can be exited by using **Exit** or **Goto**.
Form 2: Check for loop termination before executing the loop the first time.
Form 3: Execute the loop once and then check for loop termination.

Loop Termination:

Until *condexpr*: Do *statement*s until *condexpr* is **True**.

While *condexpr*: Do *statement*s while *condexpr* is **True**.

See Also

For, For Each, Exit Do**, While.**

Example

```
Sub Main
    I = 2
    Do
        I = I*2
    Loop Until I > 10
    Debug.Print I
End Sub
```

Example Output

```
16
```

DoEvents — Instruction

Syntax `DoEvents`

Description This instruction allows other applications to process events.

Example

```
Sub Main
   DoEvents          'let other apps work
End Sub
```

DropListBox — Dialog Item Definition

Syntax `DropListBox x, y, dx, dy, strarray$( ), .field _ [, Options]`

Parameters

Name	Description
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
strarray$( )	This one-dimensional array of strings establishes the list of choices. All the non-null elements of the array are used.
field	The value of the drop-down list box is accessed via this field. It is the index of the *StrArray$( )* var.
Options	If this numeric value is zero or omitted then the drop-down list is not editable. If it is non-zero then the drop-down list is also an edit box.

Description Define a drop-down listbox item.

See Also `Begin Dialog, Dim As UserDialog.`

Example

```
Sub Main
   Dim lists$(3)
   lists$(0) = "List 0"
   lists$(1) = "List 1"
   lists$(2) = "List 2"
   lists$(3) = "List 3"
   Begin Dialog UserDialog 200,120
      Text 10,10,180,15,"Please push the OK button."
```

```
      DropListBox 10,25,180,60,lists$(),
.list
      OKButton 80,90,40,20
   End Dialog
   Dim dlg As UserDialog
   dlg.list = 2
   Dialog dlg          'show Dialog (Wait For OK)
   Debug.Print dlg.list
End Sub
```

End — Instruction

Syntax

```
End
```

Description

The end instruction causes the macro to terminate immediately. If the macro was run by another macro using the **MacroRun** instruction then that macro continues on the instruction following the **MacroRun.**

Example

```
Sub DoSub
   L$ = UCase$("InputBox$ (Enter End:)")
   If L$ = "END" Then End
   Debug.Print "End was Not entered."
End Sub
Sub Main
   Debug.Print "Before DoSub"
   DoSub
   Debug.Print "After DoSub"
End Sub
```

Example Output

```
Before DoSub
End was Not entered.
After DoSub
```

Enum — Definition

Syntax

```
[ | Private | Public ] Enum name
    elem [ = value]
   [...]
End Enum
```

Description Define a new userenum. Each elem defines an element of the enum. If value is given then that is the element's value. The value can be any constant integer expression. If value is omitted then the element's value is one more than the previous element's value. If there is no previous element then zero is used.

Enum defaults to Public if neither Private or Public is specified.

Example

```
Enum Days
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
    Sunday
End Enum

Sub Main
    Dim D As Days
    For D = Monday To Friday
        Debug.Print D ' 0 through 4
    Next D
End Sub
```

Environ

Instruction/Function

Syntax **Environ[*$*](*Index*)**

-or-

Environ[*$*](*Name*)

Description Return an environment string.

Parameters

Parameter	Description
Index	Return this environment string's value. If there is no environment string at this index a null string is returned. Indexes start at one.
Name	Return this environment string's value. If the environment string can't be found a null string is returned.

Example

```
Sub Main
    Debug.Print Environ("Path")
End Sub
```

Eof — Function

Syntax **Eof(***streamnum***)**

Parameters

Name	Description
streamnum	Streams 1, 2, 3 and 4 are available in each macro.

Description Return *True* if *Streamnum* is at the end of the file.

Example

```
Sub Main
   Open XXX For Input As #1
   While Not Eof(1)
      Line Input #1,L$
      Debug.Print L$
   Wend
   Close #1
End Sub
```

Erase — Instruction

Syntax **Erase** *array*[, ...]

Description Reset *array* to zero. (Dynamic arrays are reset to undimensioned arrays.) String arrays values are set to a null string. *Array* must be declared as an array using **Dim, Private, Public** or **Static.**

Example

```
Sub Main
   Dim X%(2)
   X%(1) = 1
   Erase X%
   Debug.Print X%(1) '" 0"
End Sub
```

Example Output

```
 0
```

Err — Variable

Syntax **Err =** *errorcode*

Description Set it to zero to clear the last error condition. Use **Error** to trigger an error event. Err in an expression returns the current error code.

Example

```
Sub Main
   On Error GoTo Problem
   Error 1      'simulate Error #1
```

```
   Exit Sub
   Problem:  'Error handler
   Debug.Print "Error Number =";Err
   Debug.Print "Error String = ";Error$
   Resume Next
End Sub
```

Example Output

```
Error Number = 1
Error String = Application specific error #1.
```

Error — Instruction/Function

Syntax

Error *errorcode*
-or-
Error[$]([*errorcode*])

Parameters

Name	Description
errorcode	This is the error number.

Description

Instruction: Signal error *ErrorCode*. This triggers error handling just like a real error. The current procedures error handler is activated, unless it is already active or there isnt one. In that case the calling procedures error handler is tried. If no available error handler is found in any of the calling procedures of the current macro, the macro is halted.

Function: The Error( ) function returns the error text string.

Example

```
Sub Main
   On Error GoTo Problem
   Error 1      'simulate Error #1
   Exit Sub
   Problem:  'Error handler
   Debug.Print "Error Number =";Err
   Debug.Print "Error String = ";Error$
   Resume Next
End Sub
```

Example Output

```
Error Number = 1
Error String = Application specific error #1.
```

Exit — Instruction

Syntax

Exit {All|Do|For|Function|Property|Sub|While}

Parameters

Exit	Description
All	Exit all macros.
Do	Exit the **Do** loop.
For	Exit the **For** of **For Each** loop.
Function	Exit the Function block. Note: This instruction resets **Err** to zero and **Error$** to null.
Property	Exit the **Property** block. Note: This instruction resets **Err** to zero and **Error$** to null .
Sub	Exit the **Sub** block. Note: This instruction resets **Err** to zero and **Error$** to null.
While	Exit the **While** loop.

Description

The exit instruction causes the macro to continue without doing some or all of the remaining instructions.

Example

```
Sub DoSub(L$)
   Do
      If L$ = "DO" Then Exit Do
   I = I+1
   Loop While I < 10
   If I = 0 Then Debug.Print "Do was entered"
   For I = 1 To 10
      If L$ = "FOR" Then Exit For
   Next I
      If I = 1 Then Debug.Print "For was entered"
   I = 10
   While I > 0
      If L$ = "WHILE"Then Exit While
      I = I-1
   Wend
   If I = 10 Then Debug.Print "While was entered"
   If L$ = "SUB" Then Exit Sub
   Debug.Print "Sub was Not entered."
   If L$ = "ALL" Then Exit All
   Debug.Print "All was Not entered."
End Sub
Sub Main
   L$ = InputBox$("Enter Do, For, While,Sub Or All:")
   Debug.Print "Before DoSub"
   DoSub UCase$(L$)
   Debug.Print "After DoSub"
End Sub
```

Example Output

```
Before DoSub
Do was entered
```

```
Sub was Not entered.
All was Not entered.
After DoSub
```

Exp — Function

Syntax **Exp(***num***)**

Parameters

Name	Description
num	Return e raised to the power of this number value. The value e is approximately 2.71 8282.

Description The **Exp** function computes the exponential of the variable *num*.

Example

```
Sub Main
   Debug.Print Exp(1)
End Sub
```

Example Output

```
 2.71828182845905
```

Exp10 — Function

Syntax **Exp10(***num***)**

Parameters

Name	Description
num	Return 10 raised to the power of this number value.

Description The **Exp10** function computes the base-10 exponential of the variable *num*.

Example

```
Sub Main
   Debug.Print Exp10(1)
End Sub
```

Example Output

```
 10
```

FileAttr — Function

Syntax **FileAttr(***StreamNum, ReturnValue***)**

Description Return StreamNum's open mode or file handle.

Parameter	Description
StreamNum	Streams 1 through 255 are private to each macro. Streams 256 through 511 are shared by all macros.

ReturnValue 1 - return the mode used to open the file: 1=Input, 2=Output, 4=Random, 8=Append, 32=Binary

2 - return the file handle

See Also Open.

Example

```
Sub Main
    Open "XXX" For Output As #1
    Debug.Print FileAttr(1,1) ' 2
    Close #1
End Sub
```

FileCopy — Instruction

Syntax **FileCopy** *FromName$, ToName$*

Description Copy a file.

Parameters

Parameter	Description
FromName$	This string value is the path and name of the source file. A path relative to the current directory can be used.
ToName$	This string value is the path and name of the destination file. A path relative to the current directory can be used.

Example

```
Sub Main
    FileCopy "C:\AUTOEXEC.BAT","C:\AUTOEXEC.BAK"
End Sub
```

FileDateTime — Function

Syntax **FileDateTime(***name$***)**

Parameters

Name	Description
name$	This string value is the path and name of the file. A path relative to the current directory can be used.

Description Return the date and time file *Name$* was last changed as a *date* value. If the file does not exist then a run-time error occurs.

Example

```
Sub Main
   F$ = Dir$("*.*")
   While F$ <> ""
      Debug.Print F$;" ";"";FileDateTime(F$)
      F$ = Dir$()
   Wend
```

```
End Sub
```

Example Output

```
SNR.APB 12/22/95 4:21:06 PM
FRQ-RESP.AT1 12/22/95 4:21:06 PM
```

FileLen — Function

Syntax **FileLen(***name$***)**

Parameters

Name	Description
name$	This string value is the path and name of the file. A path relative to the current directory can be used.

Description Return the length of file *Name$*. If the file does not exist then a run-time error occurs.

Example

```
Sub Main
   F$ = Dir$("*.*")
   While F$ <> ""
      Debug.Print F$;" ";"";FileLen(F$)
      F$ = Dir$()
   Wend
End Sub
```

Example Output

```
SNR.APB  311
FRQ-RESP.AT1  31744
```

Fix — Function

Syntax **Fix(***num***)**

Parameters

Name	Description
num	Return the integer portion of this number value. The number is truncated. Positive numbers return the next lower integer. Negative numbers return the next higher integer.

Description Return the integer value.

Example

```
Sub Main
   Debug.Print Fix(9.9)
   Debug.Print Fix(0)
   Debug.Print Fix(-9.9)
End Sub
```

Example Output

```
 9
 0
-9
```

For — Statement

Syntax

```
For num = first To last [step Inc]
     statements
Next [num]
```

Parameters

Name	Description
num	This is the iteration variable.
first	Set *num* to this value initially.
last	Continue looping while *num* is in the range. See *Step* below.
step	If this number value is greater than zero then the for loop continues as long as *num* is less than or equal to *Last.* If this number value is less than zero then the for loop continues as long as *num* is greater than or equal to *Last*. If this is omitted then one is used.

Description Execute *statements* while *num* is in the range *First* to *Last.*

See Also Do, For Each, Exit For, While.

Example

```
Sub Main
   For I = 0 To 300 Step 100
      Debug.Print I;I+I;I*I
   Next I
End Sub
```

Example Output

```
 0 0 0
 100 200 10000
 200 400 40000
 300 600 90000
```

For Each — Statement

Syntax

```
For Each var In items
     statements
Next [var]
```

Parameters

Name	Description
var	This is the iteration variable.
items	This is the collection of items to be done.

Description Execute *statements* for each item in *Items.*

See Also `Do, For, Exit For, While.`

Example

```
Sub Main
   Dim Document As Object
   For Each Document In MicroSoft.Word.Documents
      Debug.Print Document.Title
   Next Document
End Sub
```

Format$ — Function

Syntax **Format**[$](*expr*[, *form$*])

Description Return the formatted string representation of *expr*.

Parameters

Name	Description
expr	Return the formatted string representation of this number value.
form	Format *expr* using to this string value. If this is omitted then return the *expr* as a string. See below: Predefined Date Format, Predefined Number Format, User defined Date Format, User defined Number Format, User defined Text Format.

Format Predefined Date

Description The following predefined date formats may be used with the **Format** function. Predefined formats may not be combined with user defined formats or other predefined formats.

Form	Description
General Date	Same as **user defined date format** "c"
Long Date	Same as **user defined date format** "dddddd"
Medium Date	Not supported at this time.
Short Date	Same as **user defined date format** "ddddd"
Long Time	Same as **user defined date format** "ttttt"
Medium Time	Same as **user defined date format** "hh:mm AMPM "
Short Time	Same as **user defined date format** "hh:mm"

Format Predefined Number

Description The following predefined number formats may be used with the **Format** function. Predefined formats may not be combined with user defined formats or other predefined formats.

Form	Description
General number	Return number as is.
Currency	Same as **user defined number format** " $#,##0.00;($#,##0.00)" **Not locale dependent at this time.**

Fixed	Same as **user defined number format** "0.00".
Standard	Same as **user defined number format** "#,##0.00".
Percent	Same as **user defined number format** "0.00%".
Scientific	Same as **user defined number format** "0.00E+00".
Yes/No	Return No if zero, else return "Yes".
True/False	Return True if zero, else return "False".
On/Off	Return On if zero, else return "Off".

Example

```
Sub Main
   Debug.Print Format$(2.145,"Standard")
End Sub
```

Example Output

```
2.15
```

Format User Defined Date

Description

The following date formats may be used with the **Format** function. Date formats may be combined to create the user defined date format. User defined date formats may not be combined with other user defined formats or predefined formats.

Form	Description
:	insert localized time separator
/	insert localized date separator
c	insert ddddd ttttt, insert date only if t=0, insert time only if d=0
d	insert day number without leading zero
dd	insert day number with leading zero
ddd	insert abbreviated day name
dddd	insert full day name
ddddd	insert date according to Short Date format
dddddd	insert date according to Long Date format
w	insert day of week number
ww	insert week of year number
m	insert month number without leading zero insert minute number without leading zero (if follows h or hh)
mm	insert month number with leading zero insert minute number with leading zero (if follows h or hh)
mmm	insert abbreviated month name
mmmm	insert full month name
q	insert quarter number
y	insert day of year number
yy	insert year number (two digits)
yyyy	insert year number (four digits, no leading zeros)
h	insert hour number without leading zero
hh	insert hour number with leading zero
n	insert minute number without leading zero
nn	insert minute number with leading zero
s	insert second number without leading zero
ss	insert second number with leading zero
ttttt	insert time according to time format

AM/PM	use 12 hour clock and insert AM (hours 0 to 11) and PM (12 to 23)
am/pm	use 12 hour clock and insert am (hours 0 to 11) and pm (12 to 23)
A/P	use 12 hour clock and insert A (hours 0 to 11) and P (12 to 23)
a/p	use 12 hour clock and insert a (hours 0 to 11) and p (12 to 23)
AMPM	use 12 hour clock and insert localized AM/PM strings
\c	insert character c
"text"	insert literal text

Format User Defined Number

Description

The following number formats may be used with the **Format** function. Number formats may be combined to create the user defined number format. User defined number formats may not be combined with other user defined formats or predefined formats.
User defined number formats can contain up to four sections separated by a semi-colon (;):

form;format for non-negative expr, -format for negative expr, empty and null expr return

form;negform - negform: format for negative expr

form;negform;zeroform - zeroform: format for zero expr

form;negform;zeroform;nullform - nullform: format for empty or null expr

Form	Description
#	digit, don't include leading/trailing zero digits (all the digits left of decimal point are returned) eg. Format(19,"###") returns "19" eg. Format(19,"#") returns "19"
0	digit, include leading/trailing zero digits eg. Format(19,"000") returns "019" eg. Format(19,"0") returns "19"
.	decimal, insert localized decimal point eg. Format(19.9,"###.00") returns "19.90" eg. Format(19.9,"###.##") returns "19.9"
,	thousands, insert localized thousand separator every 3 digits xxx, or xxx,. mean divide expr by 1000 prior to formatting two adjacent commas ",," means divide expr by 1000 again eg. Format(1900000,"0,,") returns "2" eg. Format(1900000,"0,,.0") returns "1.9"
%	percent, insert %, multiply expr by 100 prior to formatting
:	insert localized time separator
/	insert localized date separator
E+ e+ E- e-	use exponential notation, insert E (or e) and the signed exponent eg. Format(1000,"0.00E+00") returns "1.00E+03" eg. Format(.001,"0.00E+00") returns "1.00E-03"

+ $ ( )space	insert literal char eg. Format(10,"$#") returns "$10"
\c	insert character c eg. Format(19,"\####\#") returns "#19#"
"text"	insert literal text eg. Format(19,"""##""###""##""") returns "##19##"

Example

```
Sub Main
   Debug.Print Format$(2.145,"#.00")
End Sub
```

Example Output

```
2.15
```

Format User Defined Text

Description

The following text formats may be used with the **Format** function. Text formats may be combined to create the user defined text format. User defined text formats may not be combined with other user defined formats or predefined formats.
User defined text formats can contain one or two sections separated by a semi-colon (;):

form - format for all strings
form;nullform - nullform: format for null strings

Form	Description
@	char placeholder, insert char or space
&	char placeholder, insert char or nothing
<	all chars lowercase
>	all chars uppercase
!	fill placeholder from left-to-right (default is right-to-left)
\c	insert character c
"text"	insert literal text

Example

```
Sub Main
   Debug.Print Format("123","ab@c")
   Debug.Print Format("123","!ab@c")
End Sub
```

Example Output

```
12ab3c
ab1c23
```

FreeFile

Instruction

Syntax **FreeFile**[( )]

Description Return the next unused stream number. Streams 1, 2, 3 and 4 are available in each macro.

Example

```
Sub Main
   Debug.Print FreeFile  '1
   Open XXX For Input As #1
   Debug.Print FreeFile  '2
   Close #1
   Debug.Print FreeFile  '1
End Sub
```

Example Output

Function — Definition

Syntax

```
[Private|Public|Friend] Function name[type][([param[, …]])] [As type]
     statements
End Function
```

Description

User defined function. The function defines a set of statements to be executed when it is called. The values of the calling arglist are assigned to the params. Assigning to name[type] sets the value of the function result.

Function defaults to Public if Private, Public or Friend are not is specified.

See Also

Declare, Property, Sub.

Example

```
Function Power(X,Y)
   P = 1
   For I = 1 To Y
      P = P*X
   Next I
   Power = P
End Function

Sub Main
   Debug.Print Power(2,8)
End Sub
```

Example Output

```
 256
```

Get — Instruction

Syntax `Get StreamNum, [RecordNum], var`

Parameters

Name	Description
StreamNum	Streams 1 through 255 are private to each macro. Streams 256 through 511 are shared by all macros.
RecordNum	For Random mode files this is the record number. The first record is 1. Otherwise, it is the byte position. The first byte is 1. If this is omitted then the current position (or record number) is used.
var	This variable value is read from the file. For a fixed length variable (like Long) the number of bytes required to restore the variable are read. For a Variant variable two bytes are read which describe its type and then the variable value is read accordingly. For a usertype variable each field is read in sequence. For an array variable each element is read in sequence. For a dynamic array variable the number of dimensions and range of each dimension is read prior to reading the array values. All binary data values are read from the file in little-endian format.

Note: When reading a string (or a dynamic array) from a Binary mode file the length (or array dimension) information is not read. The current string length determines how much string data is read. The current array dimension determines how may array elements are read.

Description Get a variable's value from StreamNum.

See Also Open, Put.

Example

```
Sub Main
    Dim V As Variant
    Open "SAVE_V.DAT" For Binary Access Read As #1
    Get #1, , V
    Close #1
End Sub
```

GetAllSettings — Function

Syntax `GetAllSettings(AppName$, Section$, Key$)`

Parameters

Name	Description
AppName$	This string value is the name of the project which has this Section and Key.
Section$	This string value is the name of the section of the project settings.

Description Get all of Section's settings in project AppName. Settings are returned in a Variant. Empty is returned if there are no keys in the section. Otherwise, the Variant contains a two dimension array: (I,0) is the key and (I,1) is the setting.

Win16 and Win32s store settings in a .ini file named AppName. Win32 stores settings in the registration database.

Example

```
Sub Main
    SaveSetting "MyApp","Font","Size",10
    SaveSetting "MyApp","Font","Name","Courier"
    Settings = GetAllSettings("MyApp","Font")
    For I = LBound(Settings) To UBound(Settings)
        Debug.Print Settings(I,0); "="; Settings(I,1)
    Next I
    DeleteSetting "MyApp","Font"
End Sub
```

GetAttr — Function

Syntax **GetAttr(***Name$***)**

Parameters

Name	Description
Name$	This string value is the path and name of the file. A path relative to the current directory can be used.

Description Return the *attributes* for file *Name$*. If the file does not exist then a run-time error occurs.

Example

```
Sub Main
   F$ = Dir$("*.*")
   While F$ <> ""
      Debug.Print F$;"";GetAttr(F$)
      F$ = Dir$()
   Wend
End Sub
```

Example Output

```
SNR.APB 32
FRQ-RESP.AT1 32
```

GetFilePath$ — Function

Syntax **GetFilePath**[$]([*defname$*], [*defext$*], [*defdir$*], _ [*title$*], [*option*])

Parameters

Name	Description
defname$	Set the initial File Name to this string value. If this is omitted then *.*DefExt$* is used.

defext$	Initially show files whose extension matches this string value. (Multiple extensions can be specified by using ";" as the separator.) If this is omitted then * is used.
defdir$	This string value is the initial directory. If this is omitted then the current directory is used.
title$	This string value is the title of the dialog. If this is omitted then Open" is used.
option	This numeric value determines the file selection options. If this is omitted then zero is used. See table below.

Option	Effect
0	Only allow the user to select a file that exists.
1	Confirm creation when the user selects a file that does not exist.
2	Allow the user to select any file whether it exists or not.
3	Confirm overwrite when the user selects a file that exists.

Description Put up a dialog box and get a file path from the user. The returned string is a complete path and file name. If the cancel button is pressed then a null string is returned.

Example

```
Sub Main
   Debug.Print GetFilePath$("*.*")
End Sub
```

Example Output

```
C:\APWIN\Samples\S1\Snr.apb
```

GetObject — Function

Syntax **GetObject(***file$*[, *class$*]**)**

Parameters

Name	Description
filename$	This is the file where the object resides. If this is omitted then the currently active object for *Class$* is returned.
class$	This string value is the applications registered class name. If this application is not cu rrently active it will be started. If this is omitted then the application associated with the files extension will be started.

Description Get an existing object of type *Class$* from *File$*. Use **Set** to assign the returned object to an object variable.

Example

```
Sub Main
   Dim App As Object
   Set App = GetObject(,"??????.Application")
   App.Move 20,30  move icon to 20,30
   Set App = Nothing
   App.Quit  'run-time error (no object)
End Sub
```

GetSetting — Function

Syntax **GetSetting[$](***AppName$, Section$, Key$***)**

Description Get the setting for Key in Section in project AppName. Win16 and Win32s store settings in a .ini file named AppName. Win32 stores settings in the registration database.

Parameter	Description
AppName$	This string value is the name of the project which has this Section and Key.
Section$	This string value is the name of the section of the project settings.
Key$	This string value is the name of the key in the section of the project settings.

Example

```
Sub Main
    SaveSetting "MyApp","Font","Size",10
    Debug.Print GetSetting("MyApp","Font","Size") ' 10
End Sub
```

Goto — Instruction

Syntax **GoTo** *label*

Description Go to the *label* and continue execution from there. Only *labels* in the current user subroutine. Function or property are accessible.

Example

```
Sub Main
   X = 2
Label:
   X = X*X
   If X <= 100 Then GoTo Label
   Debug.Print X
End Sub
```

Example Output

```
 256
```

GroupBox Dialog Item

Definition

Syntax **GroupBox** *x, y, dx, dy, title$[, .field]*

Parameters

Name	Description
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
title$	This string value is the title of the group box.
field	This identifier is the name of the *field*. The *dialogfunc* receives this name as *string*. If this identifer is omitted then the first two words of the title are used.

Description Define a groupbox item.

See Also **Begin Dialog**, **Dim** As *UserDialog*.

Example

```
Sub Main
   Begin Dialog UserDialog 200,120
      Text 10,10,180,15,"Please push the OK button."
      GroupBox 10,25,180,60,"Group box"
      OKButton 80,90,40,20
   End Dialog
   Dim dlg As UserDialog
   Dialog dlg          'Show Dialog (Wait For OK)
End Sub
```

Hex$

Function

Syntax **Hex**[$]**(***num***)**

Parameters

Name	Description
num	Return a hex encoded string for this number value.

Description Return a hex string.

See Also Oct$( ), Str$( ), Val( ).

Example

```
Sub Main
   Debug.Print Hex$(15)
End Sub
```

Example Output F

Hour — Function

Syntax **Hour(***dateexpr***)**

Parameters

Name	Description
dateexpr	Return the hour of the day for this date value.

Description Return the hour of the day (0 to 23).

See Also Minute( ), Second( ), Time( ).

Example

```
Sub Main
   Debug.Print Hour(#12:00:01 AM#)
End Sub
```

Example Output 0

If — Statement

Syntax

```
If condexpr Then [instruction] [Else instruction]
-or-
If condexpr Then
     statements
[ElseIf condexpr Then
     statements...]
[Else
     statements]
End If
```

Description Form 1: Single line if statement. Execute the *instruction* following the Then if *condexpr* is **True**. Otherwise, execute the *instruction* following the Else. The Else portion is optional.

Form 2: The multiple line if is useful for complex ifs. Each if *condexpr* is checked in turn. The first **True** one causes the following *statements* to be executed. If all are **False** then the Elses *statements* are executed. The ElseIf and Else portions are optional.

See Also Select Case, Choose( ), IIf( ).

Example

```
Sub Main
   S = InputBox("Enter hello, goodbye, dinner Or
sleep:")
   S = UCase(S)
```

```
   If S = "HELLO" Then Debug.Print "Come In"
   If S = "GOODBYE" Then Debug.Print "See you later"
   If S = "DINNER" Then
      Debug.Print "Please come In."
      Debug.Print "Dinner will be ready soon."
   ElseIf S = "SLEEP" Then
      Debug.Print "Sorry."
      Debug.Print "We are full For the night"
   End If
End Sub
```

IIf — Function

Syntax `IIf(condexpr, truepart, falsepart)`

Parameters

Name	Description
condexpr	If this value is true then return *TruePart*. Otherwise, return *FalsePart*.
truepart	Return this value if *condexpr* is *True*.
falsepart	Return this value if *condexpr* is *False*.

Description Return the value of the indicated by *condexpr*. Both *TruePart* and *FalsePart* are evaluated.

See Also `If, Select Case, Choose( ).`

Example

```
Sub Main
   Debug.Print IIf(1 > 0,"True","False")
End Sub
```

Example Output

```
True
```

Input — Instruction

Syntax `Input [#]streamnum, var[, ...]`

Description Get input from *Streamnum* and assign it to *vars*. Input values are comma delimited. Leading and trailing spaces are ignored. If the first char (following the leading spaces) is a quote (") then the string is terminated by an ending quote. Special values #NULL#, #FALSE#, #TRUE#, #date# and #Error number# are converted to their appropriate value and data type.

See Also `Line Input, Print, Write.`

Example

```
Sub Main
   Open XXX For Input As #1
```

```
   Input #1,A,B,C$
   Debug.Print A;B;C$
   Close #1
End Sub
```

Input$ Function

Syntax **Input**[$](*n*, *streamnum*)

Parameters

Name	Description
n	Read *n* chars. If fewer than *n* chars are left before the end of file then a run-time error occurs.
streamnum	Streams 1, 2, 3 and 4 are available in each macro.

Description Return *N* chars from *Streamnum*.

Example

```
Sub Main
   Open XXX For Input As #1
   L = Lof(1)
   T$ = Input$(L,1)
   Close #1
   Debug.Print T$;
End Sub
```

InputBox$ Function

Syntax **InputBox**[$](*Prompt$*[, *title$*]
[, *default$*][, *xpos*, *ypos*])

Parameters

Name	Description
prompt$	Use this string value as the prompt in the input box.
title$	Use this string value as the title of the input box. If this is omitted then the input box does not have a title.
default$	Use this string value as the initial value in the input box. If this is omitted then the initial value is blank.
xpos	When the dialog is put up the left edge will be at this screen position. If this is omitted then the dialog will be centered.
ypos	When the dialog is put up the top edge will be at this screen position. If this is omitted then the dialog will be centered.

Description Display an input box where the user can enter a line of text. Pressing the OK button returns the string entered. Pressing the Cancel button returns a null string.

Example

```
Sub Main
```

```
   L$ = InputBox$("Enter some Text:","Input Box
⇨Example","Example text")
   Debug.Print L$
End Sub
```

Example Output

```
Example text
```

InStr — Function

Syntax `InStr([index, ]String1$, String2$)`

Parameters

Name	Description
index	Start searching for *S2$* at this offset in *S1$*. If this is omitted then start searching from the beginning of *S1$*.
string1$	Search for *S2$* in this string value.
string2$	Search *S1$* for this string value.

Description Return the index where *S2$* first matches *S1$*. If no match is found return 0.

See Also Left$( ), Len( ), Mid$( ), Right$( ).

Example

```
Sub Main
   Debug.Print InStr("Hello","l")
End Sub
```

Example Output

```
 3
```

InStrRev — Function

Syntax `InStrRev(S1$, S2$[, Index])`

Description Return the index where S2$ last matches S1$. If no match is found return 0.

Parameters

Name	Description
S1$	Search for S2$ in this string value. If this value is Null then Null is returned.
S2$	Search S1$ for this string value. If this value is Null then Null is returned.
Index	Start searching for S2$ ending at this index in S1$. If this is omitted then start searching from the end of S1$.

See Also Left$( ), Len( ), Mid$( ), Replace$( ), Right$( ).

Example

```
Sub Main
    Debug.Print InStrRev("Hello","l") ' 4
End Sub
```

Int — Function

Syntax `Int(num)`

Parameters

Name	Description
num	Return the largest integer which is less than or equal to this number value.

Description Return the integer value.

Example

```
Sub Main
   Debug.Print Int(9.9)
   Debug.Print Int(0)
   Debug.Print Int(-9.9)
End Sub
```

Example Output

```
 9
 0
-10
```

Is — Operator

Syntax *expr* ***Is*** *expr*

Description Return the *True* if both *exprs* refer to the same object.

See Also `Objects.`

Example

```
Sub Main
   Dim X As Object
   Dim Y As Object
   Debug.Print X Is Y
End Sub
```

Example Output

```
True
```

IsArray — Function

Syntax `IsArray(var)`

Parameters

Name	Description
var	A array variable or a variant var can contain multiple values.

Description Return the *True* if *var* is an array of values.

See Also TypeName, VarType.

Example

```
Sub Main
   Dim X As Variant, Y(2) As Integer
   Debug.Print IsArray(X)
   X = Array(1,4,9)
   Debug.Print IsArray(X)
   X = Y
   Debug.Print IsArray(X)
End Sub
```

Example Output

```
False
True
True
```

IsDate — Function

Syntax **IsDate(***expr***)**

Parameters

Name	Description
expr	A variant expression to test for a valid date.

Description Return the *True* if *expr* is a valid date.

See Also TypeName, VarType.

Example

```
Sub Main
   Dim X As Variant
   X = 1
   Debug.Print IsDate(X)
   X = Now
   Debug.Print IsDate(X)
End Sub
```

Example Output

```
False
True
```

IsEmpty — Function

Syntax **IsEmpty(***variantvar***)**

Parameters

Name	Description
variantvar	A variant var is *Empty* if it has never been assigned a value.

Description Return the *True* if *variantvar* is *Empty*.

See Also TypeName, VarType.

Example

```
Sub Main
   Dim X As Variant
   Debug.Print IsEmpty(X)
   X = 0
   Debug.Print IsEmpty(X)
   X = Empty
   Debug.Print IsEmpty(X)
End Sub
```

Example Output

```
True
False
True
```

IsError — Function

Syntax **IsError(***expr***)**

Parameters

Name	Description
expr	A variant expression to test for an error code value.

Description Return the *True* if *expr* is an error code.

See Also TypeName, VarType.

Example

```
Sub Main
   Dim X As Variant
   Debug.Print IsError(X)
   X = CVErr(1)
   Debug.Print IsError(X)
End Sub
```

Example Output

```
False
True
```

IsMissing Function

Syntax **IsMissing(***variantvar***)**

Parameters

Name	Description
variantvar	Return True if this parameters argument expression was not specified in the **Sub, Function** or **Property** call.

Description Return the *True* if Optional parameter *variantvar* did not get a value. An Optional or ParamArray parameter may be omitted in the **Sub, Function** or **Property** call.

Example

```
Sub Main
    Opt              'IsMissing(A)=True
    Opt "Hi"          'IsMissing(A)=False
    Many             'No args
    Many 1,"Hello"   'A(0)=1 A(1)=Hello
End Sub
Sub Opt(Optional A)
    Debug.Print "IsMissing(A) = ";IsMissing(A)
End Sub
Sub Many(ParamArray A())
    If LBound(A) > UBound(A) Then
        Debug.Print "No args"
    Else
        For I = LBound(A) To UBound(A)
            Debug.Print "A(" & I & ") = " & A(I) & " "
        Next I
        Debug.Print
    End If
End Sub
```

Example Output

```
IsMissing(A) = True
IsMissing(A) = False
No args
A(0) = 1
A(1) = Hello
```

IsNull Function

Syntax **IsNull(***expr***)**

Parameters

Name	Description
expr	A variant expression to test for *Null.*

Description Return the *True* if *expr* is *Null.*

See Also `TypeName, VarType.`

Example

```
Sub Main
    Dim X As Variant
    Debug.Print IsNull(X) '(IsEmpty, but not IsNull)
    X = 1
    Debug.Print IsNull(X)
    X = "1"
    Debug.Print IsNull(X)
    X = Null
    Debug.Print IsNull(X)
    X = X*2
    Debug.Print IsNull(X)
End Sub
```

Example Output

```
False
False
False
True
True
```

IsNumeric — Function

Syntax **IsNumeric(***expr***)**

Parameters

Name	Description
expr	A variant expression is a numeric value if it is *numeric* or string value that represents a number.

Description Return the *True* if *expr* is a numeric value.

See Also `TypeName, VarType.`

Example

```
Sub Main
   Dim X As Variant
   X = 1
   Debug.Print IsNumeric(X)
   X = "1"
   Debug.Print IsNumeric(X)
   X = "A"
   Debug.Print IsNumeric(X)
End Sub
```

Example Output

```
True
```

```
True
False
```

IsObject — Function

Syntax `IsObject(var)`

Parameters	Name	Description
	var	A var contains an object reference if it is *objexpr* reference.

Description Return the *True* if *var* contains an object reference.

See Also `TypeName, VarType.`

Example

```
Sub Main
   Dim X As Variant
   X = 1
   Debug.Print IsObject(X)
   X = 1"
   Debug.Print IsObject(X)
   Set X = Nothing
   Debug.Print IsObject(X)
End Sub
```

Example Output

```
False
False
True
```

Kill — Instruction

Syntax `Kill Name$`

Parameters	Name	Description
	name$	This string value is the path and name of the file. A path relative to the current directory can be used.

Description Delete the file named by *name$*.

Example

```
Sub Main
   Kill "FILENAME.EXT"
End Sub
```

LBound — Function

Syntax `LBound(var[, dimension])`

Parameters

Name	Description
var	Return the lowest index for this array variable.
dimension	Return the lowest index for this dimension of *var*. If this is omitted then return the lowest index for the first dimension.

Description Return the lowest index.

See Also `UBound( ).`

Example

```
Sub Main
   Dim A(-1 To 3,2 To 6)
   Debug.Print LBound(A)
   Debug.Print LBound(A,1)
   Debug.Print LBound(A,2)
End Sub
```

Example Output

```
-1
-1
 2
```

LCase$ — Function

Syntax `LCase[$](string$)`

Parameters

Name	Description
string$	Return the string value of this after all chars have been converted to lowercase.

Description Return a string from *string$* where all the uppercase letters have been lowercased.

See Also `UCase$( ).`

Example

```
Sub Main
   Debug.Print LCase$("Hello")
End Sub
```

Example Output

```
hello
```

Left$ Function

Syntax `Left[$](string$, len)`

Parameters

Name	Description
string$	Return the left portion of this string value.
len	Return this many chars. If *string$* is shorter than that then just return *string$*.

Description Return a string from *S$* with only the *Len* chars.

See Also `InStr( ), Len( ), Mid$( ), Right$( ).`

Example

```
Sub Main
   Debug.Print Left$("Hello",2)
End Sub
```

Example Output

```
He
```

Len Function

Syntax `Len(string$)`

Parameters

Name	Description
string$	Return the number of chars in this string value.

Description Return the number of characters in *string$*.

See Also `InStr( ), Left$( ), Mid$( ), Right$( ).`

Example

```
Sub Main
   Debug.Print Len("Hello")
End Sub
```

Example Output

```
5
```

Let Instruction

Syntax `[Let] var = expr`

Description Assign the value of *expr* to *var*. The keyword Let is optional.

Example

```
Sub Main
   Let X = 1
   X = X*2
   Debug.Print X
End Sub
```

Example Output 2

Like — Operator

Syntax `str1 Like str2`

Description Return the True if str1 matches pattern str2. The pattern in str2 is one or more of the special character sequences shown in the following table.

Char(s)	Description
?	Match any single character.
***	Match zero or more characters.
#	Match a single digit (0-9).
[charlist]	Match any char in the list.
[!charlist]	Match any char not in the list.

Example

```
Sub Main
    Dim X As Object
    Dim Y As Object
    Debug.Print X Is Y ' True
End Sub
```

Line Input — Instruction

Syntax **Line Input** [#]*streamnum*, *string$*

Description Get a line of input from *Streamnum* and assign it to *string$*.

See Also Input, Print, Write.

Example

```
Sub Main
   Open "FILENAME.EXT" For Input As #1
   Line Input #1,S$
   Debug.Print S$
   Close #1
End Sub
```

Example Output

ListBox Dialog Item — Definition

Syntax **ListBox** *x, y, dx, dy, strarray$( ), .field*

Parameters

Name	Description
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
strarray$( )	This one-dimensional array of strings establishes the list of choices. All the non-null elements of the array are used.
field	The value of the list box is accessed via this *field*. It is the index of the *StrArray$( )* var.

Description Define a listbox item.

See Also `Begin Dialog, Dim As UserDialog.`

Example

```
Sub Main
    Dim lists$(3)
    lists$(0) = "List 0"
    lists$(1) = "List 1"
    lists$(2) = "List 2"
    lists$(3) = "List 3"
    Begin Dialog UserDialog 200,120
        Text 10,10,180,15,"Please push the OK button"
        ListBox 10,25,180,60,lists$(),.list
        OKButton 80,90,40,20
    End Dialog
    Dim dlg As UserDialog
    dlg.list = 2
    Dialog dlg ' show dialog (wait for ok)
    Debug.Print dlg.list
End Sub
```

Example Output

Loc — Function

Syntax **Loc(***streamnum***)**

Parameters

Name	Description
streamnum	Streams 1, 2, 3 and 4 are available in each macro.

Description Return *Streamnum* file position.

Example

```
Sub Main
   Open "FILENAME.EXE" For Input As #1
   L = Loc(1)
   Close #1
   Debug.Print L
End Sub
```

Example Output

```
1
```

Lock

Instruction

Syntax

Lock *StreamNum*

-or-

Lock *StreamNum*, *RecordNum*

-or-

Lock *StreamNum*, [*start*] **To** *end*

Parameters

Name	Description
StreamNum	Streams 1 through 255 are private to each macro. Streams 256 through 511 are shared by all macros.
RecordNum	For Random mode files this is the record number. The first record is 1. Otherwise, it is the byte position. The first byte is 1.
start	First record (or byte) in the range.
end	Last record (or byte) in the range.

Description

Form 1: Lock all of StreamNum.

Form 2: Lock a record (or byte) of StreamNum.

Form 3: Lock a range of records (or bytes) of StreamNum. If start is omitted then lock starting at the first record (or byte).

Note: Be sure to Unlock for each Lock instruction.

Note: For sequential files (Input, Output and Append) lock always affects the entire file.

See Also

Open, Unlock.

Example

```
Sub Main
    Dim V As Variant
    Open "SAVE_V.DAT" For Binary As #1
    Lock #1
    Get #1, 1, V
    V = "Hello"
    Put #1, 1, V
```

```
    Unlock #1
    Close #1
End Sub
```

LOF — Function

Syntax **Lof(***streamnum***)**

Parameters

Name	Description
streamnum	Streams 1 through 255 are private to each macro. Streams 256 through 511 are shared by all macros.

Description Return StreamNum file length (in bytes).

Example

```
Sub Main
   Open "FILENAME.EXT" For Input As #1
   L = Lof(1)
   Close #1
   Debug.Print L
End Sub
```

Example Output Length of file value.

Log — Function

Syntax **Log(***num***)**

Parameters

Name	Description
num	Return the natural logarithm of this number value. The value e is approximately 2.718282.

Description Return the natural logarithm.

Example

```
Sub Main
   Debug.Print Log(1)
End Sub
```

Example Output

```
0
```

Log10 — Function

Syntax **Log10(***num***)**

Parameters

Name	Description
num	Return the base-10 logarithm of this number value.

Description Return the base-10 logarithm.

Example

```
Sub Main
   Debug.Print Log10(24)
End Sub
```

Example Output

```
 1.38021124171161
```

LSet — Instruction

Syntax

```
LSet strvar = str
-or-
LSet usertypevar1 = usertypevar2
```

Description Form 1: Assign the value of *str* to *strvar*. Shorten str by removing trailing chars (or extend with blanks). The previous length *strvar* is maintained.

Form 2: Assign the value of *usertypevar2* to *usertypevar1*. If *usertypevar2* is longer than *usertypevar1* then only copy as much as *usertypevar1* can handle.

See Also RSet.

Example

```
Sub Main
    S$ = "123"
    LSet S$ = "A"
    Debug.Print ".";S$;"."
End Sub
```

Example Output

```
.A  .
```

LTrim$ — Function

Syntax

```
LTrim[$](string$)
```

Parameters

Name	Description
string$	Copy this string without the leading spaces.

Description Return the string with *string$*s leading spaces removed.

See Also Trim$( ), RTrim$( ).

Example

```
Sub Main
    Debug.Print ".";LTrim$("  x  ");"."
End Sub
```

Example Output

```
.x  .
```

MacroDir$ Function

Syntax **MacroDir**[$]

Description Return the directory of the current macro. A run-time error occurs if the current macro has never been saved.

See Also MacroRun.

Example

```
Sub Main
   ' Open the file called Data that is in the
   ' same directory as the macro
   Open MacroDir & "\Data" For Input As #1
   Line Input #1, S$
   Close #1
End Sub
```

MacroRun Instruction

Syntax **MacroRun** *command$*

Parameters

Name	Description
command$	Start the macro named by this string value. That macros **Command$** is assigned the text following first space in this value.

Description Play a macro. Execution will continue at the following statement after the macro has completed.

See Also Command$.

Example

```
Sub Main
    Debug.Print "Before Demo"
    MacroRun "APDEMO.APB"
    Debug.Print "After Demo"
End Sub
```

MacroRunThis Instruction

Syntax MacroRunThis MacroCode$

Description Play the macro code. Execution will continue at the following statement after the macro code has completed. The macro code can be either a single line or a complete macro.

Parameter	Description
MacroName$	Run the macro named by this string value.

See Also Command$, MacroDir$, MacroRun.

Example

```
Sub Main
    Debug.Print "Before Demo"
    MacroRunThis "MsgBox ""Hello"""
    Debug.Print "After Demo"
End Sub
```

Main

Sub

Syntax

```
Sub Main()
    ...
End Sub
```

-or-

```
Private Sub Main()
    ...
End Sub
```

Description Form 1: Each macro must define Sub Main. A macro is a "program". Running a macro starts the Sub Main and continues to execute until the subroutine finishes.

Form 2: A code module may define a Private Sub Main. This Sub Main is the code module initialization subroutine. If Main is not defined then no special initialization occurs.

See Also Code Module.

Me

Object

Syntax `Me`

Description Me references the current macro/module. It can be used like any other object variable, except that it's reference can't be changed.

See Also `Set`.

Example

```
Sub Main
    DoIt
```

```
    Me.DoIt ' calls the same sub
End Sub
Sub DoIt
    MsgBox "Hello"
End Sub
```

Mid$ Function/Assignment

Syntax

Mid[$](*string$, index*[, *len*])
-or-
Mid[$](*strvar, index*[, *len*]) **=** *string$*

Parameters

Name	Description (Mid Function)
string$	Copy chars from this string value.
index	Start copying chars starting at this index value. If the string is not that long then return a null string.
len	Copy this many chars. If the *string$* does not have that many chars starting at *Index* then copy the remainder of *string$.*

Name	Description (Mid Assignment)
strvar	Change part of this string.
index	Change *strvar* starting at this index value. If the string is not that long then it is not changed.
len	The number of chars copied is smallest of: the value of *Len*, the length of *string$* and the remaining length of *strvar.* (If this value is omitted then the number of chars copied is the smallest of: the length of *string$* and the remaining length of *strvar.*)
string$	Copy chars from this string value.

Description

Function: Return the substring of S$ starting at *Index* for *Len* chars.

Instruction: Assign *string$* to the substring in *strvar* starting at *Index* for *Len* chars.

Example

```
Sub Main
    S$ = "Hello There"
    Mid$(S$,7) = "??????????"
    Debug.Print S$ '"Hello ?????"
    Debug.Print Mid$("Hello",2,1)
End Sub
```

Example Output

```
Hello ?????
e
```

Minute — Function

Syntax `Minute(dateexpr)`

Parameters

Name	Description
dateexpr	Return the minute of the hour for this date value.

Description Return the minute of the hour (0 to 59).

See Also `Hour( ), Second( ), Time( ).`

Example

```
Sub Main
    Debug.Print Minute(#12:15:01 AM#)
End Sub
```

Example Output

```
 15
```

MkDir — Instruction

Syntax `MkDir` *name$*

Parameters

Name	Description
name$	This string value is the path and name of the directory. A path relative to the current directory can be used.

Description Make directory *name$*.

See Also `RmDir.`

Example

```
Sub Main
   MkDir "C:\APTEMP"
End Sub
```

Month — Function

Syntax `Month(dateexpr)`

Parameters

Name	Description
dateexpr	Return the month of the year for this date value.

Description Return the month of the year (1 to 12).

See Also `Date( ), Day( ), Weekday( ), Year( ).`

Example

```
Sub Main
   Debug.Print Month(#1/1/1900#)
End Sub
```

Example Output

```
 1
```

MonthName — Function

Syntax

```
MonthNamw(NumZ{day}[, CondZ{abbrev}])
```

Parameters

Name	Description
day	Return the month of the year for this date value.
abbrev	If this conditional value is True then return the abbreviated form of the month name.

Description

Return the localized name of the month.

See Also

`Month( ).`

Example

```
Sub Main
    Debug.Print MonthName(1) 'January
    Debug.Print MonthName(Month(Now))
End Sub
```

MsgBox — Instruction/Function

Syntax

```
MsgBox message$[, type][, title$]
-or-
MsgBox(message$[, type][, title$])
```

Parameters

Name	Description
message$	This string value is the text that is shown in the message box.
type	This number value controls the type of message box. See the table below.
title$	This string value is the title of the message box.

Category Effect (result)	Type	
Buttons	0	OK(1) button
	1	OK(1) and Cancel(2) buttons
	2	Abort(3), Retry(4), Ignore(5) buttons
	3	Yes(6), No(7), Cancel(2) buttons
	4	Yes(6) and No(7) buttons
	5	Retry(4) and Cancel(2) buttons
Icons	0	No icon
	16	Stop icon
	32	Question icon
	48	Attention icon
	64	Information icon
Default	0	First button
	256	Second button
	512	Third button
Mode	0	Application modal
	4096	System modal

Description Show a message box titled *Title$*. *Type* controls what the message box looks like (choose one value from each category). Use MsgBox( ) if you need to know what button was pressed. The result indicates which button was pressed.

Example

```
Sub Main
    If MsgBox("Please press OK button",1) = 1 Then
        Debug.Print "OK was pressed"
    Else
        Debug.Print "Cancel was pressed"
    End If
End Sub
```

Name — Instruction

Syntax **Name** *oldname$* **As** *newname$*

Parameters

Name	Description
oldname$	This string value is the path and name of the file. A path relative to the current directory can be used.
newname$	This is the new file name. The file remains in its original directory.

Description Rename file *oldname$* as *newname$*.

Example

```
Sub Main
   Name "AUTOEXEC.BAK" As "AUTOEXEC.SAV"
End Sub
```

Now — Function

Syntax **Now**

Description Return the current date and time as a *date* value.

See Also Date, Time, Timer.

Example

```
Sub Main
   Debug.Print Now
End Sub
```

Example Output

```
2/9/96 7:59:26 AM
```

Oct$ — Function

Syntax **Oct**[$]**(***num***)**

Parameters

Name	Description
num	Return an octal encoded string for this number value.

Description Return a octal string.

See Also `Hex$( ), Str$( ), Val( ).`

Example

```
Sub Main
   Debug.Print Oct$(15)
End Sub
```

Example Output

```
17
```

Object — Module

Description (The Object module feature is not implemented in version 1.5 of AP Basic)

An object module implements an OLE Automation object.

- It has a set of Public properties, functions and subroutines accessible from other macros and modules.
- These public symbols are accessed via the name of the object module or an object variable.
- Public Consts, Types, arrays, fixed length strings are not allowed.
- An object module is similar to a class module except that one instance is automatically created. That instance has the same name as the object module's name.
- To create additional instances use:

```
Dim Obj As objectname
Set Obj = New objectname
```

See Also Class Module, Code Module, Uses.

Example

```
'A.WWB
'#Uses "System.OBM"
Sub Main
    Debug.Print Hex(System.Version)
End Sub

'System.OBM
Option Explicit
Declare Function GetVersion16 Lib "Kernel" _
    Alias "GetVersion" () As Long
Declare Function GetVersion32 Lib "Kernel32" _
```

```
    Alias "GetVersion" () As Long

Public Function Version() As Long
    If Win16 Then
        Version = GetVersion16
    Else
        Version = GetVersion32
    End If
End Function
```

Object_Initialize Sub

Syntax

```
Private Sub Object_Initialize()
    ...
End Sub
```

Description

Object module initialization subroutine. Each time a new instance is created for a Object module the Object_Initialize sub is called. If Object_Initialize is not defined then no special initialization occurs.

Note: Object_Initialize is also called for the instance that is automatically created.

See Also

Object Module, Object_Terminate.

Object_Terminate Sub

Syntax

```
Private Sub Object_Terminate()
    ...
End Sub
```

Description

Object module termination subroutine. Each time an instance is destroyed for a Object module the Object_Terminate sub is called. If Object_Terminate is not defined then no special termination occurs.

See Also

Object Module, Object_Initialize.

Oct$ — Function

Syntax `Oct[$](Num)`

Description Return a octal string.

Parameter	Description
Num	Return an octal encoded string for this number value.

See Also Hex$( ), Str$( ), Val( ).

Example

```
Sub Main
    Debug.Print Oct$(15) '17
End Sub
```

OKButton Dialog Item — Definition

Syntax `OKButton x, y, dx, dy[, .field]`

Parameters

Name	Description
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
field	This identifier is the name of the *field*. The *dialogfunc* receives this name as *string*. If this identifer is omitted then the first two words of the title are used. If this is omitted then the field name is OK.

Description Define an OK button item. Pressing the OK button updates the *dlgvar* field values and closes the dialog. (**Dialog**( ) function call returns -1.)

See Also `Begin Dialog, Dim As UserDialog.`

Example

```
Sub Main
    Begin Dialog UserDialog 200,120
        Text 10,10,180,30,"Please push the OK button"
        OKButton 80,90,40,20
    End Dialog
    Dim dlg As UserDialog
    Dialog dlg 'Show dialog (Wait for OK)
End Sub
```

On Error — Instruction

Syntax

```
On Error GoTo 0
-or-
On Error GoTo label
-or-
On Error Resume Next
```

Description

Form 1: Disable the error handler (default).

Form 2: Send error conditions to an error handler.

Form 3: Error conditions continue execution at the next statement.

On Error sets or disables the error handler. Each user defined subroutine, function or property has its own error handler. The default is to terminate the macro on any error. The **Err** variable is set whenever an error occurs. Once an error has occurred and the error handler is executing any further errors will terminate the macro, unless **Err** has been set to zero.

Note: This instruction resets **Err** to zero and **Error$** to null.

Example

```
Sub Main
    On Error Resume Next
    Error 1
    Debug.Print "RESUMING, Err=";Err
    On Error GoTo X
    Error 1
    Exit Sub

X:  Debug.Print "Err=";Err
    Err = 0
    Resume Next
End Sub
```

Example Output

```
RESUMING, Err= 1
Err= 1
```

Open — Instruction

Syntax

```
Open name$ For mode As [#]streamnum
```

Parameters

Name	Description
name$	This string value is the path and name of the file. A path relative to the current directory can be used.
mode	May be Input, Output or Append.
streamnum	Streams 1, 2, 3 and 4 are available in each macro.

Description Open file *Name$* for mode as *Streamnum*.

See Also `Close, Reset.`

Example

```
Sub Main
Open "FILENAME.EXT" For Output As #1
   Print #1,"1,2,""Hello"""
   Close #1
End Sub
```

Option

Definition

Syntax **`Option Explicit`**

Description Require all variables to be declared prior to use. Variables are declared using **Dim**, **Private** or **Public** or **Static**.

See Also Option Explicit

Example

```
Option Explicit
Sub Main
    Dim A
    A = 1
    B = 2   'B has not been declared.
End Sub
```

OptionButton Dialog Item

Definition

Syntax **`OptionButton`** *x, y, dx, dy, title$*[, *.field*]

Parameters

Name	Description
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
title$	The value of this string is the title of the option button.

Description Define an option button item.

See Also `Begin Dialog, Dim As` *`UserDialog`*`, OptionGroup.`

Example

```
Sub Main
    Begin Dialog UserDialog 200,120
        Text 10,10,180,15,"Please push the OK button."
        OptionGroup .options
            OptionButton 10,30,180,15,"Option &0"
            OptionButton 10,45,180,15,"Option &1"
            OptionButton 10,60,180,15,"Option &2"
        OKButton 80,90,40,20
    End Dialog
    Dim dlg As UserDialog
    dlg.options = 2
    Dialog dlg           'Show dialog (Wait for OK)
    Debug.Print dlg.options
End Sub
```

OptionGroup

Dialog Item Definition

Syntax

OptionGroup *.field*

OptionButton *x, y, dx, dy, title$*[, *.field*]

OptionButton *x, y, dx, dy, title$*[, *.field*]

...

Parameters

Name	Description
field	The value of the option group is accessed via this field. This first option button is 0, the second is 1, etc.
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
title$	The value of this string is the title of the option button.

Description Define a optiongroup and option button items.

See Also Begin Dialog, Dim As UserDialog, OptionButton.

Example

```
Sub Main
    Begin Dialog UserDialog 200,120
        Text 10,10,180,15,"Please push the OK button."
```

```
        OptionGroup .options
            OptionButton 10,30,180,15,"Option &0"
            OptionButton 10,45,180,15,"Option &1"
            OptionButton 10,60,180,15,"Option &2"
        OKButton 80,90,40,20
    End Dialog
    Dim dlg As UserDialog
    dlg.options = 2
    Dialog dlg        'Show dialog (Wait for OK)
    Debug.Print dlg.options
End Sub
```

Pow — Function

Syntax **Pow(***numx, powery***)**

Parameters

Name	Description
numx	Number X to be rased.
powery	Power of Y.

Description Return the value of a number (NumX) raised to the power of (PowerY).

Example

```
Sub Main
   Debug.Print Pow(3,3)
End Sub
```

Example Output

```
 27
```

Picture Dialog Item — Definition

Syntax **Picture** *X, Y, DX, DY, FileName$, Type*[, *.Field*]

Description Define a picture item. The bitmap is automatically sized to fit the item's entire area.

Parameter	Description
X	This number value is the distance from the left edge of the dialog box. It is measured in 1/8 ths of the average character width for the dialog's font.
Y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12 ths of the character height for the dialog's font.
DX	This number value is the width. It is measured in 1/8 ths of the average character width for the dialog's font.

DY	This number value is the height. It is measured in 1/12 ths of the character height for the dialog's font.
FileName$	The value of this string is the .BMP file shown in the picture control.
Type	This numeric value indicates the type of bitmap used. See below.
Field	This identifier is the name of the field. The dialogfunc receives this name as string. If this identifer is omitted then the first two words of the title are used.

Type	Effect
0	FileName is the name of the bitmap file. If the file does not exist then "(missing picture)" is displayed.
3	The clipboard's bitmap is displayed. Not supported.
+16	Instead of displaying "(missing picture)" a run-time error occurs.

See Also Begin Dialog, Dim As UserDialog.

Example

```
Sub Main
    Begin Dialog UserDialog 200,120
        Picture 10,10,180,75,"SAMPLE.BMP",0
        OKButton 80,90,40,20
    End Dialog
    Dim dlg As UserDialog
    Dialog dlg ' show dialog (wait for ok)
End Sub
```

PowerRatioTodB — Function

Syntax **PowerRatioTodB(***num***)**

Parameters

Name	Description
num	

Description Return the value in dB of the power ratio of *num* to 1.

Example

```
Sub Main
   Debug.Print Format(PowerRatioTodB(.5), "#.0000")
End Sub
```

Example Output

```
 -3.0103
```

Equation

```
PowerRatioTodB = 10 * Log10(Num)
```

Print — Instruction

Syntax **Print #***streamnum*, [*expr*[; ...][;]]

Description Print the *expr*(s) to *Streamnum*. Use ; to separate expressions. A *num* is automatically converted to a string before printing (just like **Str$**( )). If the instruction does not end with a ; then a newline is printed at the end.

See Also `Input, Line Input, Write.`

Example

```
Sub Main
   A = 1
   B = 2
   C$ = Hello
   Open "FILENAME.EXT" For Output As #1
   Print #1,A;",";B;",""";C$;"""
   Close #1
End Sub
```

Private — Definition

Syntax ***Private*** *name*[*type*][([Dim[, …]])] [As *type*][, ...]

Description Create arrays (or simple variables) which are available to the entire macro, but not other macros. Dimension var array(s) using the *dimlist* to establish the minimum and maximum index value for each dimension. If the *dims* is omitted then a scalar (single value) variable is defined. A dynamic array is declared using ( ) without any *dims*. It must be **ReDim**ensioned before it can be used. The Private statement must be placed outside of **Sub**, **Function** or **Property** blocks.

See Also `Dim, Public, ReDim, Static.`

Example

```
Private A0,A1(1),A2(1,1)
Sub Init
   A0 = 1
   A1(0) = 2
   A2(0,0) = 3
End Sub
Sub Main
   Init
   Debug.Print A0;A1(0);A2(0,0)
End Sub
```

Example Output

```
 1 2 3
```

Private — Keyword

Description Private **Consts, Declares, Functions, Privates, Propertys, Subs** and **Types** are only available in the current macro.

Property — Definition

Syntax

```
[Private|Public] Property Get name[type][([param[, _
     ]])] [As type]
     statements
End Property
```

-or-

```
[Private|Public] Property [LetSet] name[([param[, _
     ]])]
     statements
End Property
```

Description User defined property. The property defines a set of *statements* to be executed when its value is used or changed. A property acts like a variable, except that getting its value calls Property Get and changing its value calls Property Let (or Property Set). Property Get and Property Let with the same *name* define a property that holds a value. Property Get and Property Set with the same *name* define a property that holds an object reference. The values of the calling *arglist* are assigned to the parameters in the *params.*

For Property Let and Property Set the last parameter is the value on the right hand side of the assignment operator.

Public is assumed if neither *Private* or *Public* is specified.

See Also Function, Sub.

Example

```
Dim X_Value
Property Get X()
    X = X_Value
End Property
Property Let X(NewValue)
    If Not IsNull(NewValue) Then X_Value = NewValue
End Property

Sub Main
    X = "Hello"
    Debug.Print X
    X = Null
```

```
    Debug.Print X
End Sub
```

Example Output

```
Hello
Null
```

Public — Definition

Syntax ***Public*** *name*[*type*][([Dim[, …]])] [As *type*][, ...]

Description Create arrays (or simple variables) which are available to the entire macro and other macros. Dimension var array(s) using the *dims* to establish the minimum and maximum index value for each dimension. If the *dims* are omitted then a scalar (single value) variable is defined. A dynamic array is declared using ( ) without any *dims*. It must be **ReDim**ensioned before it can be used. The Public statement must be placed outside of **Sub**, **Function** or **Property** blocks.

See Also Dim, Private, ReDim, Static.

Example

```
Public A0,A1(1),A2(1,1)
   Sub Init
   A0 = 1
   A1(0) = 2
   A2(0,0) = 3
End Sub
Sub Main
   Init
   Debug.Print A0;A1(0);A2(0,0)
End Sub
```

Example Output

```
 1 2 3
```

Public — Keyword

Description Public **Consts**, **Declares, Functions, Propertys, Publics, Subs** and **Types** in hidden macros are available in all other macros.

PushButton Dialog Item — Definition

Syntax **PushButton** *x*, *y*, *dx*, *dy*, *title$*[, *.field*]

Parameters

Name	Description
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
title$	The value of this string is the title of the push button control.
field	This identifier is the name of the field. The *dialogfunc* receives this name as *string*. If this identifer is omitted then the first two words of the title are used.

Description Define a push button item. Pressing the push button updates the *dlgvar* field values and closes the dialog. (**Dialog**( ) function call returns the push buttons ordinal number in the dialog. The first push button returns 1.)

See Also `Begin Dialog, Dim As UserDialog.`

Example

```
Sub Main
    Begin Dialog UserDialog 200,120
        Text 10,10,180,30,"Please push the DoIt button"
        OKButton 40,90,40,20
        PushButton 110,90,60,20,"&Do It"
    End Dialog
    Dim dlg As UserDialog
    Debug.Print Dialog(dlg)
End Sub
```

Put — Instruction

Syntax **Put** *StreamNum*, [*RecordNum*], *var*

Parameters

Name	Description
StreamNum	Streams 1 through 255 are private to each macro. Streams 256 through 511 are shared by all macros.
RecordNum	For Random mode files this is the record number. The first record is 1. Otherwise, it is the byte position. The first byte is 1. If this is omitted then the current position (or record number) is used.
var	This variable value is written to the file. For a fixed length variable (like Long) the number of bytes required to store the variable are written. For a Variant variable two bytes which describe its type are written and then the variable value is written accordingly. For a usertype variable each field is written in sequence. For an array

variable each element is written in sequence. For a dynamic array variable the number of dimensions and range of each dimension is written prior to writing the array values. All binary data values are written to the file in little-endian format.

Note: When a writing string (or a dynamic array) to a Binary mode file the string length (or array dimension) information is not written. Only the string data or array elements are written.

Description Write a variable's value to StreamNum.

See Also `Get, Open.`

Example

```
Sub Main
    Dim V As Variant
    Open "SAVE_V.DAT" For Binary Access Write As #1
    Put #1, , V
    Close #1
End Sub
```

QBColor — Function

Syntax **`QBColor(`***num***`)`**

Parameters

num	color
0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	yellow
7	white
8	gray
9	light blue
10	light green
11	light cyan
12	light red
13	light magenta
14	light yellow
15	bright white

Description Return the appropriate color defined by Quick Basic.

See Also `RGB( ).`

Example

```
Sub Main
   Debug.Print Hex(QBColor(1))
   Debug.Print Hex(QBColor(7))
```

```
    Debug.Print Hex(QBColor(8))
    Debug.Print Hex(QBColor(9))
    Debug.Print Hex(QBColor(10))
    Debug.Print Hex(QBColor(12))
    Debug.Print Hex(QBColor(15))
End Sub
```

Example Output

```
800000
C4C4C4
808080
FF0000
FF00
FF
FFFFFF
```

Randomize — Instruction

Syntax `Randomize`

Description Randomize the random number generator.

See Also `Rnd( ).`

Example

```
Sub Main
   Randomize
   Debug.Print Rnd
End Sub
```

Example Output

```
 0.84881130405591
```

ReDim — Instruction

Syntax `ReDim [Preserve] name[type][([Dim[, …]])] [As _ type][, ...]`

Description Redimension a dynamic array. Use Preserve to keep the array values. Otherwise, the array values will all be reset. When using **Preserve** only the last index of the array may change. The number of indexes may not. (A one-dimensional array cant be redimensioned as a two-dimensional array.)

See Also `Dim, Private, Public, Static.`

Example

```
Sub Main
   Dim X()
   ReDim X(3)
   Debug.Print UBound(X)
   ReDim X(200)
   Debug.Print UBound(X)
End Sub
```

Example Output

```
 3
 200
```

Reference — Comment

Syntax

```
'#Reference
{uuid}#vermajor.verminor#lcid#[path[#name]]
```

Description The Reference comment indicates that the current macro/module references the type library identified. Reference comment lines must be the first lines in the macro/module (following the global Attributes). Reference comments are in reverse priority (from lowest to highest). The IDE does not display the reference comments.

Parameters

Name	Description
uuid	Type library's universally unique identifier.
vermajor	Type library's major version number.
verminor	Type library's minor version number.
lcid	Type library's locale identifier.
path	Type library's path.
name	Type library's name.

Example

```
'#Reference
{00025E01-0000-0000-C000-000000000046}#4.0#0#C: _
\PROGRAM FILES\COMMON FILES\MICROSOFT SHARED\DAO\_
DAO350.DLL#Microsoft DAO 3.5 Object Library
```

Rem — Instruction

Syntax

```
Rem ...
-or-
'...
```

Description Both forms are comments. The Rem form is an instruction. The form can be used at the end of any macro line. All text from either " ' " or Rem to the end of the line is part of the comment. That text is not executed.

Example

```
Sub Main
    Debug.Print "Hello" 'Prints to the output window.
    Rem the macro terminates at Main's End Sub
End Sub
```

Example Output

```
Hello
```

Replace — Function

Syntax **Replace[$](***S***, *Pat*, Rep, [*Index*], [*Count*])**

Description Replace Pat with Rep in S.

Parameters

Name	Description
S	This string value is searched. Replacements are made in the string returned by Replace.
Pat	This string value is the pattern to look for.
Rep	This string value is the replacement.

Index	This numeric value is the starting index in S. Replace(S,Pat,Rep,N) is equivalent to Replace(Mid(S,N),Pat,Rep). If this is omitted use 1.
Count	This numeric value is the maximum number of replacements that will be done. If this is omitted use -1 (which means replace all occurrences).

See Also `InStr( ), InStrRev( ), Left$( ), Len( ), Mid$( ), Right$( ).`

Example

```
Sub Main
    Debug.Print Replace$("abcabc","b","B")     '"aBcaBc"
    Debug.Print Replace$("abcabc","b","B",,1) '"aBcabc"
    Debug.Print Replace$("abcabc","b","B",3)   '"caBc"
    Debug.Print Replace$("abcabc","b","B",9)   '""
End Sub
```

Reset — Instruction

Syntax **Reset**

Description Close all open streams for the current macro.

See Also `Close, Open.`

Example

```
Sub Main
    ' Read the first line of XXX and print it.
    Open "FILENAME.EXT" For Input As #1
    Line Input #1,L$
    Debug.Print L$
    Reset
End Sub
```

Resume — Instruction

Syntax

Resume *label*
-or-
Resume Next

Description Form 1: Resume execution at *label*.

Form 2: Resume execution at the next statement.

Once an error has occurred, the error handler can use Resume to continue execution. The error handler must use Resume or **Exit** at the end. Executing an End **Sub** (or End **Function**) while in an error handler causes a run-time error.

Note: This instruction resets **Err** to zero and **Error$** to null.

Example

```
Sub Main
    On Error GoTo X
    Error 1
    Debug.Print "RESUMING"
    Exit Sub

X:  Debug.Print "Err=";Err
    Resume Next
End Sub
```

Example Output

```
RESUMING
```

RGB

Function

Syntax **RGB(***red, green, blue***)**

Description Return a color.

See Also QBColor( ).

Example

```
Sub Main
   Debug.Print Hex(RGB(255,0,0))
End Sub
```

Example Output FF

Right$

Function

Syntax **Right**[$]**(***string$, len***)**

Parameters

Name	Description
string$	Return the right portion of this string value.
len	Return this many chars. If *string$* is shorter than that then just return *string$*.

Description Return the last Len chars of *string$*.

See Also InStr( ), Left$( ), Len( ), Mid$( ).

Example

```
Sub Main
   Debug.Print Right$(Hello,3)
End Sub
```

Example Output

```
Llo
```

RmDir — Instruction

Syntax `RmDir name$`

Parameters

Name	Description
name$	This string value is the path and name of the directory. A path relative to the current directory can be used.

Description Remove directory Name$.

See Also `MkDir`.

Example

```
Sub Main
   RmDir "C:\APTEMP"
End Sub
```

Rnd — Function

Syntax `Rnd([num])`

Parameters

Name	Description
num	This number value is ignored.

Description Return a random number greater than or equal to zero and less than one.

See Also `Randomize`.

Example

```
Sub Main
   Debug.Print Rnd()
End Sub
```

Example Output

```
 0.95883053071688
```

Round — Function

Syntax **Round**([*Num*][, *Places*])

Parameters

Name	Description
num	Round this numeric value. If this value is Null then Null is returned.
Places	Round to this number of decimal places. If this is omitted then round to the nearest integer value.

Description Return the number rounded to the specified number of decimal places.

Example

```
Sub Main
    Debug.Print Round(.5)        ' 0
    Debug.Print Round(.500001)   ' 1
    Debug.Print Round(1.499999)  ' 1
    Debug.Print Round(1.5)       ' 2
    Debug.Print Round(11.11)     ' 11
    Debug.Print Round(11.11,1)   ' 11.1
End Sub
```

RSet — Instruction

Syntax **RSet** *strvar* **=** *str*

Description Assign the value of *str* to *strvar*. Shorten *str* by removing trailing chars (or extend with leading blanks). The previous length *strvar* is maintained.

See Also LSet.

Example

```
Sub Main
    S$ = "123"
    RSet S$ = "A"
    Debug.Print ".";S$;"."
End Sub
```

Example Output

```
.  A.
```

RTrim$ — Function

Syntax **RTrim**[$](*string$*)

Parameters

Name	Description
string$	Copy this string without the trailing spaces.

Description Return the string with *string$*s trailing spaces removed.

See Also `LTrim$( ), Trim$( ).`

Example

```
Sub Main
    Debug.Print ".";RTrim$("  x  ");"."
End Sub
```

Example Output

```
.  x.
```

SaveSetting — Instruction

Syntax **SaveSetting** *AppName$, Section$, Key$, Setting*

Description Save the Setting for Key in Section in project AppName. Win16 and Win32s store settings in a .ini file named AppName. Win32 stores settings in the registration database.

Parameter	Description
AppName$	This string value is the name of the project which has this Section and Key.
Section$	This string value is the name of the section of the project settings.
Key$	This string value is the name of the key in the section of the project settings.
Setting	Set the key to this value. (The value is stored as a string.)

Example

```
Sub Main
    SaveSetting "MyApp","Font","Size",10
End Sub
```

Second — Function

Syntax **Second(***dateexpr***)**

Parameters

Name	Description
dateexpr	Return the second of the minute for this date value.

Description Return the second of the minute (0 to 59).

See Also `Hour( ), Minute( ), Time( ).`

Example

```
Sub Main
    Debug.Print Second(#12:00:01 AM#)
End Sub
```

Example Output

```
 1
```

Seek — Instruction

Syntax

```
Seek [#]streamnum, count
```

Parameters

Name	Description
streamnum	Streams 1, 2, 3 and 4 are available in each macro.
count	This number value is the number of bytes to skip over from the beginning of the file.

Description Position *Streamnum* for input *Count*.

See Also `Seek( ).`

Example

```
Sub Main
    Open "FILEMANE.EXT" For Input As #1
    Line Input #1,L$
    Seek #1,0 ' Rewind to start of file.
    Input #1,A
    Close #1
    Debug.Print A
End Sub
```

Seek — Function

Syntax

```
Seek(streamnum)
```

Parameters

Name	Description
streamnum	Streams 1, 2, 3 and 4 are available in each macro.

Description Return StreamNum current position.

See Also `Seek.`

Example

```
Sub Main
    Open "FILENAME.EXT" For Input As #1
    Line Input #1,L$
    Debug.Print Seek(1)
    Close #1
End Sub
```

Select Case — Statement

Syntax

```
Select Case expr
     Case caseexpr[, ...]
          statements
```

```
    [Case Else
        statements]
End Select
```

Parameters

caseexpr	Description
expr	Execute if equal.
Is < expr	Execute if less than.
Is <= expr	Execute if less than or equal to.
Is > expr	Execute if greater than.
Is >= expr	Execute if greater than or equal to.
Is <> expr	Execute if not equal to.
expr1 To expr2	Execute if greater than or equal to *expr1* and less than or equal to *expr2*.

Description

Select the appropriate case by comparing the *expr* with each of the caseexprs. Select the Case Else part if no caseexpr matches. (If the Case Else is omitted then skip the entire Select...End Select block.)

See Also

`If, Choose( ), IIf( ).`

Example

```
Sub Main
    S$ = InputBox$("Enter hello, goodbye, dinner or sleep:")
    Select Case UCase$(S$)
    Case "HELLO"
        Debug.Print "come in"
    Case "GOODBYE"
        Debug.Print "see you later"
    Case "DINNER"
        Debug.Print "Please come in."
        Debug.Print "Dinner will be ready soon."
    Case "SLEEP"
        Debug.Print "Sorry."
        Debug.Print "We are full for the night"
    Case Else
        Debug.Print "What?"
    End Select
End Sub
```

Example Output

SendKeys — Instruction

Syntax

SendKeys *keys$*[, *wait*]

Parameters

Name	Description
keys$	Send the keys in this string value to Windows.
wait	If this is not zero then the keys are sent before executing the next instruction. If this is omitted or zero then the keys are sent during the following instructions.

Keys$	Description
+	Shift modifier key: the following key is a shifted key
^	Ctrl modifier key: the following key is a control key
%	Alt modifier key: the following key is an alt key
~	Enter key
(keys)	Modifiers apply to all keys
{special n}	special key (n is an optional repeat count)
k	k Key (k is any single char)
K	Shift k Key (K is any capital letter)

Description

Send *Keys$* to Windows.

Special Keys:

Key	Description
k	k Key (any single char)
Cancel	Break Key
Esc or Escape	Escape Key
Enter	Enter Key
Menu	Menu Key (Alt)
Help	Help Key (?)
Prtsc	Print Screen Key
Print	?
Select	?
Execute	?
Tab	Tab Key
Pause	Pause Key
BS, BkSp or BackSpace	Back Space Key
Del or Delete	Delete Key
Ins or Insert	Insert Key
K	shift k Key
Left	Left Arrow Key
Right	Right Arrow Key
Up	Up Arrow Key
Down	Down Arrow Key
PgUp	Page Up Key
PgDn	Page Down Key
Home	Home Key
End	End Key
Clear	Num Pad 5 Key
Pad0 to Pad9	Num Pad 0 to 9 Keys
*Pad**	Num Pad * Key

Pad+	Num Pad + Key
PadEnter	Num Pad Enter Key
Pad-	Num Pad - Key
Pad.	Num Pad . Key
Pad/	Num Pad / Key
F1 to F24	F1 to F24 Keys

See Also AppActivate, Shell( ).

Example

```
Sub Main
    SendKeys "%S"      ' send Alt-S (Search)
    SendKeys "GoTo~~" ' send G o T o {Enter} {Enter}
End Sub
```

Set — Instruction

Syntax

Set *objvar* **=** *objexpr*
-or-
Set *objvar* **= New** *objtype*

Description Form 1: Set *objvar*s object reference to the object reference of *objexpr.*

Form 2: Set *objvars* object reference to the a new instance of *cotype* (a component object type.)

The Set instruction is how object references are assigned.

Example

```
Sub Main
   Dim Excel As Object
   Set Excel = CreateObject("Excel.Application")
End Sub
```

SetAttr — Instruction

Syntax **SetAttr** *name$, attrib*

Parameters

Name	Description
name$	This string value is the path and name of the file. A path relative to the current directory can be used.
attrib	Set the files *attributes* to this numeric value.

Description Set the *attributes* for file *Name$.* If the file does not exist then a run-time error occurs.

Example

```
Sub Main
    Attrib = GetAttr("FILENAME.EXT")
    SetAttr "FILENAME.EXE",1    'Readonly
```

```
    Debug.Print GetAttr("FILENAME.EXE")
    SetAttr "FILENAME.EXE",Attrib
End Sub
```

Example Output

```
 1
```

Sgn — Function

Syntax `Sgn(num)`

Parameters

Name	Description
num	Return the sign of this number value. Return -1 for negative. Return 0 for zero. Return 1 for positive.

Description Return the sign.

Example

```
Sub Main
   Debug.Print Sgn(9)
   Debug.Print Sgn(0)
   Debug.Print Sgn(-9)
End Sub
```

Example Output

```
 1
 0
-1
```

Shell — Function

Syntax `Shell(name$[, windowtype])`

Parameters

Name	Description
name$	This string value is the path and name of the program to run. Command line arguments follow the program name. (A long file name containing a space must be surrounded by literal double quotes.)
windowtype	This controls how the applications main window is shown. See the table below.

WindowType	Effect
1, 5, 9	Normal Window
2	Minimized Window (default)
3	Maximized Window
4, 8	Normal Deactivated Window
6, 7	Minimized Deactivated Window

Description Execute program *Name$.* This is the same as using File|Run from the Program Manager. This instruction can run .COM, .EXE, .BAT and .PIF files. If successful, return the task ID.

See Also `AppActivate, SendKeys.`

Example

```
Sub Main
    X = Shell("Calc",4)  'Run the calc program.
    AppActivate "Calculator"
    SendKeys "10{+}30*2=",1 '70
End Sub
```

Sin — Function

Syntax **`Sin(`***`num`***`)`**

Parameters

Name	Description
num	Return the sine of this number value. This is the number of radians. There are 2*Pi radians in a full circle.

Description Return the sine.

Example

```
Sub Main
   Debug.Print Sin(1)
End Sub
```

Example Output

```
 0.841470984807897
```

Space$ — Function

Syntax **`Space`**`[$]`**`(`***`len`***`)`**

Parameters

Name	Description
len	Create a string this many spaces long.

Description Return the string *Len* spaces long.

See Also `String$( ).`

Example

```
Sub Main
    Debug.Print "."; Space$(3); "."
End Sub
```

Example Output

```
.    .
```

Sqr — Function

Syntax `Sqr(num)`

Parameters

Name	Description
num	Return the square root of this number value.

Description Return the square root.

Example

```
Sub Main
   Debug.Print Sqr(9)
End Sub
```

Example Output

```
3
```

Static — Definition

Syntax `Static name[type][([Dim[, …]])] [As type][, ...]`

Description A static variable retains it value between procedure calls. Dimension var array(s) using the *dims* to establish the minimum and maximum index value for each dimension. If the *dims* is omitted then a scalar (single value) variable is defined. A dynamic array is declared using ( ) without any *dims*. It must be **ReDim**ensioned before it can be used.

See Also `Dim, Private, Public, ReDim.`

Example

```
Sub A
    Static X
    Debug.Print X
    X = "Hello"
End Sub

Sub Main
    A
    A ' prints "Hello"
End Sub
```

Example Output

```
Hello
```

Stop — Instruction

Syntax `Stop`

Description Pause macro execution. If execution is resumed then it starts at the next instruction. Use **End** to terminate the macro completely.

Example

```
Sub Main
    For I = 1 To 10
        Debug.Print I
        If I = 3 Then Stop
    Next I
End Sub
```

Example Output

```
 1
 2
 3
```

Str$ Function

Syntax **Str**[$]**(***num***)**

Parameters

Name	Description
Len	Return the string representation of this number value. Positive values begin with a blank. Negative values begin with a dash -.

Description Return the string representation of *num*.

See Also CStr( ), Hex$( ), Oct$( ), Val( ).

Example

```
Sub Main
   Debug.Print Str$(9*9)
End Sub
```

Example Output

```
81
```

StrComp$ Function

Syntax **StrComp(***Str1*,*Str2*,*Comp***)**

Description Compare two strings.

Parameter	Description
Str1	Compare this string with Str2. If this value is Null then Null is returned.
Str2	Compare this string with Str1. If this value is Null then Null is returned.
Comp	This numeric value indicates the type of comparison. If this is omitted or zero then binary comparison is used. Otherwise, text comparison is used. (Text comparison is not case sensitive.)

Result	Description
-1	Str1 is less than Str2.

0	Str1 is equal to Str2.
1	Str1 is greater than Str2.
Null	Str1 or Str2 is Null.

See Also LCase$( ), StrConv$( ), UCase$( ).

Example

```
Sub Main
    Debug.Print StrComp("F","e") ' -1
    Debug.Print StrComp("F","e",1) ' 1
    Debug.Print StrComp("F","f",1) ' 0
End Sub
```

StrConv$ Function

Syntax **StrConv[***$***](***Str,Conv***)**

Description Convert the string.

Parameter	Description
Str	Convert this string value. If this value is Null then Null is returned.
Conv	This numeric value indicates the type of conversion. See conversion table below.

Conv Value	Effect		
vbUpperCase		*1*	Convert to upper case.
vbLowerCase		*2*	Convert to lower case.
vbProperCase		*3*	Convert to proper case. (Not supported.)
vbWide		*4*	Convert to wide. (Only supported for Win32 in eastern locales.)
vbNarrow	*8*		Convert to narrow. (Only supported for Win32 in eastern locales.)
vbKatakana		*16*	Convert to Katakana. (Only supported for Win32 in Japanese locales.)
vbHiragana		*32*	Convert to Hiragana. (Only supported for Win32 in Japanese locales.)
vbUnicode	*64*		Convert to Unicode. (Only supported for Win32.)
vbFromUnicode	*128*		Convert from Unicode. (Only supported for Win32.)

See Also LCase$( ), StrComp( ), UCase$( ).

Example

```
Sub Main
    Dim B(1 To 3) As Byte
```

```
    B(1) = 65
    B(2) = 66
    B(3) = 67
    Debug.Print StrConv$(B,vbUnicode) '"ABC"
End Sub
```

StrReverse$ Function

Syntax **String**[*$*](*S*)

Parameters

Name	Description
S	Return this string with the characters in reverse order.

Description Return the string with the characters in reverse order.

Example

```
Sub Main
    Debug.Print StrReverse$("ABC") 'CBA
End Sub
```

String$ Function

Syntax **String**[$]**(***len, CHAR|$***)**

Parameters

Name	Description
len	Create a string this many chars long.
char|$	Fill the string with this char value. If this is a number value then use the ASCII char equivalent. If this is a string value use the first char of that string.

Description Return the string *Len* long filled with *Char* or the first char of *Char$*.

See Also Space$( ).

Example

```
Sub Main
   Debug.Print String$(4,65)
   Debug.Print String$(4,"ABC")
End Sub
```

Example Output

```
AAAA
AAAA
```

Sub

Definition

Syntax

```
[Private|Public] Sub name[([param[, …]])]
     statements
End Sub
```

Description

User defined subroutine. The subroutine defines a set of *statements* to be executed when it is called. The values of the calling *arglist* are assigned to the *params*. A subroutine does not return a result. Every macro has at least one subroutine. Sub Main must be defined. The macros execution begins at Sub Main. Sub Main must not have any *params*.

Public is assumed if neither *Private* or *Public* is specified.

See Also

Declare, Function, Property.

Example

```
Sub IdentityArray(A()) ' A() is an array of numbers
    For I = LBound(A) To UBound(A)
        A(I) = I
    Next I
End Sub

Sub CalcArray(A(),B,C) ' A() is an array of numbers
    For I = LBound(A) To UBound(A)
        A(I) = A(I)*B+C
    Next I
End Sub

Sub ShowArray(A()) ' A() is an array of numbers
    For I = LBound(A) To UBound(A)
        Debug.Print "(";I;")=";A(I)
    Next I
End Sub

Sub Main
    Dim X(1 To 4)
    IdentityArray X() ' X(1)=1, X(2)=2, X(3)=3, X(4)=4
    CalcArray X(),2,3 ' X(1)=5, X(2)=7, X(3)=9, X(4)=11
    ShowArray X()     ' print X(1), X(2), X(3), X(4)
End Sub
```

Example Output

```
( 1)= 5
( 2)= 7
( 3)= 9
( 4)= 11
```

Tan

Function

Syntax **Tan(***num***)**

Parameters

Name	Description
num	Return the tangent of this number value.

Description Return the tangent.

Example

```
Sub Main
   Debug.Print Tan(1)
End Sub
```

Example Output

```
1.5574077246549
```

Text Dialog Item

Definition

Syntax **Text** *x, y, dx, dy,* title$[, .field]

Parameters

Name	Description
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measure d in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
title$	The value of this string is the title of the text control.
field	This identifier is the name of the field. The *dialogfunc* receives this name as *string*. If this identifer is omitted then the first two words of the title are used.

Description Define a text item.

See Also Begin Dialog, Dim As *UserDialog*.

Example

```
Sub Main
    Begin Dialog UserDialog 200,120
        Text 10,10,180,15,"Please push the OK button."
        OKButton 80,90,40,20
    End Dialog
    Dim dlg As UserDialog
    Dialog dlg        'Show dialog (Wait for OK)
End Sub
```

TextBox Dialog Item

Definition

Syntax **TextBox** *x*, *y*, *dx*, *dy*, .field$[, options]

Parameters

Name	Description
x	This number value is the distance from the left edge of the dialog box. It is measured in 1/8ths of the average character width for the dialog's font.
y	This number value is the distance from the top edge of the dialog box. It is measured in 1/12ths of the character height for the dialog's font.
dx	This number value is the width. It is measured in 1/8ths of the average character width for the dialog's font.
dy	This number value is the height. It is measured in 1/12ths of the character height for the dialog's font.
field	The value of the text box is accessed via this field.
options	If this numeric value is zero or omitted then a single line of text can be entered. If it is less than zero then a hidden password can be entered. If it is greater than zero then multiple lines of text can be entered.

Description Define a textbox item.

See Also Begin Dialog, Dim As *UserDialog*.

Example

```
Sub Main
    Begin Dialog UserDialog 200,120
        Text 10,10,180,15,"Please push the OK button"
        TextBox 10,25,180,20,.Text$
        OKButton 80,90,40,20
    End Dialog
    Dim dlg As UserDialog
    dlg.Text$ = "none"
    Dialog dlg ' show dialog (wait for ok)
    Debug.Print dlg.Text$
End Sub
```

Time

Function

Syntax **Time**[$]

Description Return the current time as a *date* value.

See Also Date, Now, Timer.

Example

```
Sub Main
   Debug.Print Time
```

```
End Sub
```

Example Output

```
12:04:25 PM
```

Timer — Function

Syntax `Timer`

Description Return the number of seconds past midnight. (This is a real number, accurate to about 1/18th of a second.)

See Also `Date, Now, Time.`

Example

```
Sub Main
   Debug.Print Timer
End Sub
```

Example Output

```
 45284.53
```

TimeSerial — Function

Syntax **`TimeSerial(`***`hour, minute, second`***`)`**

Parameters

Name	Description
hour	This numeric value is the hour (0 to 23).
minute	This numeric value is the minute (0 to 59).
second	This numeric value is the second (0 to 59).

Description Return a *date* value.

See Also `DateSerial, DateValue, TimeValue.`

Example

```
Sub Main
   Debug.Print TimeSerial(13,30,0)
End Sub
```

Example Output

```
1:30:00 PM
```

TimeValue — Function

Syntax **`TimeValue(`***`date$`***`)`**

Parameters

Name	Description
date$	Convert this string value to the time part of date it represents.

Description Return the time part of date encoded as a string value.

See Also `DateSerial, DateValue, TimeSerial.`

Example

```
Sub Main
    Debug.Print TimeValue("1/1/2000 12:00:01 AM")
End Sub
```

Example Output

```
12:00:01 AM
```

Trim$ — Function

Syntax **Trim**[$](*string$*)

Parameters

Name	Description
string$	Copy this string without the leading or trailing spaces.

Description Return the string with S*$*s leading and trailing spaces removed.

See Also LTrim$( ), RTrim$( ).

Example

```
Sub Main
    Debug.Print ".";Trim$("  x  ");"."
End Sub
```

Example Output

```
.x.
```

Type — Definition

Syntax

```
[Private|Public] Type name
     elem [(Dim[, …])] As type[...]
End Type
```

Description Define a new *usertype*. Each *elem* defines an element of the type for storing data. As *type* defines the type of data that can be stored. A *User-defined type variable* has a value for each *elem*. Use *.elem* to access individual element values.

Public is assumed if neither *Private* or *Public* is specified.

Example

```
Type Employee
    Name As String
    Title As String
    Salary As Double
End Type

Sub Main
    Dim e As Employee
    e.Name = "John Doe"
    e.Title = "President"
```

```
    e.Salary = 100000
    Debug.Print e.Name   '"John Doe"
    Debug.Print e.Title  '"President"
    Debug.Print e.Salary ' 100000
End Sub
```

Example Output

```
John Doe
President
100000
```

TypeName — Function

Syntax **TypeName**[$](*var*)

Parameters

Name	Description
var	Return a string indicating the type of value stored in this variable.

Result

Value	Description
Empty	*Variant* variable is empty. It has never been assigned a value.
Null	*Variant* variable is null.
Integer	Variable contains an *integer* value.
Long	Variable contains a *long* value.
Single	Variable contains a *single* value.
Double	Variable contains a *double* value.
Currency	Variable contains a *currency* value.
Date	Variable contains a *date* value.
String	Variable contains a *string* value.
Object	Variable contains a *object* reference that is not Nothing. (An object may return a type name specific to that type of object.)
Nothing	Variable contains a *object* reference that is Nothing.
Error	Variable contains a error code value.
Boolean	Variable contains a *boolean* value.
Variant	Variable contains a variant value. (Only used for arrays of variants.)
Unknown	Variable contains a non-OLE Automation object reference.
Byte	Variable contains a byte value.
( )	Variable contains an array value. The TypeName of the element followed by ( ).

Description Return a string indicating the type of value stored in *var*.

See Also VarType.

Example

```
Sub Main
    Dim X As Variant
```

```
    Debug.Print TypeName(X)
    X = 1
    Debug.Print TypeName(X)
    X = 100000
    Debug.Print TypeName(X)
    X = 1.1
    Debug.Print TypeName(X)
    X = "A"
    Debug.Print TypeName(X)
    Set X = CreateObject("Word.Basic")
    Debug.Print TypeName(X)
    X = Empty
    X = Array(0,1,2)
    Debug.Print TypeName(X)
End Sub
```

Example Output

```
Empty
Integer
Long
Double
String
wordbasic
Variant()
```

UBound — Function

Syntax `UBound(var[, dimension])`

Parameters

Name	Description
var	Return the highest index for this array variable.
dimension	Return the highest index for this dimension of *var*. If this is omitted then return the highest index for the first dimension.

Description Return the highest index.

See Also `LBound( ).`

Example

```
Sub Main
   Dim A(3,6)
   Debug.Print UBound(A)
   Debug.Print UBound(A,1)
   Debug.Print UBound(A,2)
End Sub
```

Example Output

```
3
3
6
```

UCase$ Function

Syntax **UCase**[$](*string$*)

Parameters

Name	Description
string$	Return string value after all chars have been converted to uppercase.

Description Return a string from S*$* where all the lowercase letters have been uppercased.

See Also LCase$( ).

Example

```
Sub Main
   Debug.Print UCase$("Hello")
End Sub
```

Example Output

```
HELLO
```

Unlock Instruction

Syntax

Unlock *StreamNum*

-or-

Unlock *StreamNum*, *RecordNum*

-or-

Unlock *StreamNum*, *[start]* **To** *end*

Parameters

Name	Description
StreamNum	Streams 1 through 255 are private to each macro. Streams 256 through 511 are shared by all macros.
RecordNum	For Random mode files this is the record number. The first record is 1. Otherwise, it is the byte position. The first byte is 1.
start	First record (or byte) in the range.
end	Last record (or byte) in the range.

Description Form 1: Unlock all of StreamNum.

Form 2: Unlock a record (or byte) of StreamNum.

Form 3: Unlock a range of records (or bytes) of StreamNum. If start is omitted then unlock starting at the first record (or byte).

Note: For sequential files (Input, Output and Append) unlock always affects the entire file.

See Also `Lock, Open.`

Example

```
Sub Main
    Dim V As Variant
    Open "SAVE_V.DAT" For Binary As #1
    Lock #1
    Get #1, 1, V
    V = "Hello"
    Put #1, 1, V
    Unlock #1
    Close #1
End Sub
```

Uses — Comment

Syntax '#Uses "module"

-or-

'$Include: "module"

Description The Uses comment indicates that the current macro/module uses public symbols from the module.

See Also Class Module, Code Module, Object Module.

Example

```
'Macro A.WWB
'#Uses "B.WWB"
Sub Main
    Debug.Print BFunc$("Hello") '"HELLO"
End Sub

'Module B.WWB
Public Function BFunc$(S$)
    BFunc$ = UCase(S$)
End Sub
```

Val

Function

Syntax **Val(***string$***)**

Parameters

Name	Description
string$	Return the number value for this string value. A string value beginning with &O is an octal number. A string value beginning with &H is a hex number. Otherwise it is decimal number.

Description Return the value of the *string$*.

Example

```
Sub Main
   Debug.Print Val("-1000")
End Sub
```

Example Output

```
-1000
```

VarType

Function

Syntax **VarType(***var***)**

Parameters

Name	Description
var	Return a number indicating the type of value stored in this variable.

Result

Value	Description
0	*Variant* variable is empty. It has never been assigned a value.
1	*Variant* variable is null.
2	Variable contains an *integer* value.
3	Variable contains a *long* value.
4	Variable contains a *single* value.
5	Variable contains a *double* value.
6	Variable contains a *currency* value.
7	Variable contains a *date* value.
8	Variable contains a *string* value.
9	Variable contains a *object* reference.
10	Variable contains a error code value.
11	Variable contains a *boolean* value.
12	Variable contains a variant value. (Only used for arrays of variants.)
13	Variable contains a non-OLE Automation object reference.
17	Variable contains a byte value.
+8192	Variable contains an array value. Use VarType( ) And 255 to get the type of element stored in the array.

Description Return a number indicating the type of value stored in *var*.

See Also TypeName.

Example

```
Sub Main
    Dim X As Variant
    Debug.Print VarType(X)
    X = 1
    Debug.Print VarType(X)
    X = 100000
    Debug.Print VarType(X)
    X = 1.1
    Debug.Print VarType(X)
    X = "A"
    Debug.Print VarType(X)
    Set X = CreateObject("Word.Basic")
    Debug.Print VarType(X)
    X = Empty
    X = Array(0,1,2)
    Debug.Print VarType(X)
End Sub
```

Example Output

```
 0
 2
 3
 5
 8
 9
 8204
```

VoltageRatioTodB — Function

Syntax `VoltageRatioTodB(num)`

Parameters

Name	Description
num	

Description Return the value in dB of the voltage ratio of *num* to 1.

Example

```
Sub Main
   Debug.Print Format(VoltageRatioTodB(2), "#.0000")
Sub
```

Example Output

```
 6.0206
```

Equation `VoltageRatio = 20 * Log10(num)`

Wait — Function

Syntax **Wait** *Delay*

Description Wait for *Delay* seconds.

Example

```
Sub Main
   Wait 5  'Wait for 5 seconds.
End Sub
```

WaitAndDoEvents — Instruction

Syntax **WaitAndDoEvents** Delay

Description Wait for Delay seconds while giving other events on the computer time to continue. This is the preferred over Wait if any other activity needs to be kept running efficiently (such as APWIN sweeps). Because other events are kept running, timing will be slightly less accurate than if Wait is used.

See Also Wait.

Example

```
Sub Main
   WaitAndDoEvents 5 ' wait for 5 seconds
End Sub
```

Weekday — Function

Syntax **Weekday(***dateexpr***)**

Parameters

Name	Description
dateexpr	Return the weekday for this date value.

Description Return the weekday (1 to 7). Sunday=1, Monday=2, Tuesday=3, Wednesday=4, Thursday=5, Friday=6 and Saturday=7.

See Also Date( ), Day( ), Month( ), Year( ).

Example

```
Sub Main
    Debug.Print Weekday(#1/1/1996#)
End Sub
```

Example Output

```
2
```

WeekdayName — Function

Syntax **WeekdayNamw**(NumZ{*day*}[, CondZ{*abbrev*}])

Parameters

Name	Description
day	Return the month of the year for this date value.
abbrev	If this conditional value is True then return the abbreviated form of the month name.

Description Return the localized name of the weekday.

See Also Month( ).

Example

```
Sub Main
    Debug.Print WeekdayName(1) 'Sunday
    Debug.Print WeekdayName(Weekday(Now))
End Sub
```

While — Statement

Syntax

```
While condexpr
     statements
Wend
```

Description Execute *statements* while *condexpr* is **True**.

See Also Do, For, For Each, Exit While.

Example

```
Sub Main
    I = 2
    While I < 10
        I = I*2
    Wend
    Debug.Print I
End Sub
```

Example Output

```
16
```

With — Statement

Syntax

```
With objexpr
     statements
End With
```

Description *Method* and *property* references may be abbreviated inside a With block. Use *.method* or *.property* to access the object specified by the With *objexpr*.

Example

```
Sub Main
   Dim Excel As Object
   Set Excel = CreateObject("Excel.Application")
   With Excel
      Excel.Visible = True
      Excel.Quit
   End With
   Set Excel = Nothing
End Sub
```

WithEvents — Definition

Syntax

```
[Dim | Private | Public] _
WithEvents name As objtype[, ...]
```

Description Dimensioning a module level variable WithEvents allows the macro to implement event handling Subs. The variable's As type must be a type from a referenced type library (or language extension) which implements events.

Remarks This keyword is supported by the single DLL IDE/interpreter (aka the Enterprise edition). It is not supported by the interpreter implemented in WW_CU516.DLL or WW_CU532.DLL.

See Also Dim, Private, Public.

Example

```
Dim WithEvents X As Thing
Sub Main
    Set X = New Thing
    X.DoIt ' DoIt method raises DoingIt event
End Sub
Private Sub X_DoingIt
    Debug.Print "X.DoingIt event"
End Sub
```

Write — Instruction

Syntax **Write** #*streamnum*, *expr*[, ...]

Description Writes *expr*(s) to *Streamnum*. String values are quoted. Null values are written as #NULL#. Boolean values are written as #FALSE# or #TRUE#. Date values are written as #date#. Error codes are written as #Error number#.

See Also Input, Line Input, Print.

Example

```
Sub Main
```

```
    A = 1
    B = 2
    C$ = "Hello"
    Open "FILENAME.EXT" For Output As #1
    Write #1,A,B,C$
    Close #1
End Sub
```

Year — Function

Syntax **Year(***dateexpr***)**

Parameters

Name	Description
dateexpr	Return the year for this date value.

Description Return the year.

See Also Date( ), Day( ), Month( ), Weekday( ).

Example

```
Sub Main
   Debug.Print Year(#1/1/1996#)
End Sub
```

Example Output

```
 1996
```

User Notes

Appendix A

Terms

arglist [|*expr*|*param*:=*expr*][, ...]

A list of zero or more *exprs* that are assigned to the parameters of the sub, function or property.

- A positional parameter may be skipped by omitting the expression. Only optional parameters may be skipped.
- Positional parameter assignment is done with *expr*. Each parameter is assigned in turn. By name parameter assignment may follow.
- By name parameter assignment is done with *param*:=*expr*. All following parameters must be assigned by name.

As [New] type Dim, Private, Public and Static statements may declare variable types using As type or As New objtype. A variable declared using As New objtype is automatically created prior to use, if the variable is Nothing.

As type Variable and argument types, as well as, function and property results may be specified using As type: *Boolean*, *Byte, Currency*, *Date*, Double, *Integer*, *Long*, *Object, Single, String, String*n, UserDialog, Variant, usertype*.

attribute A file attribute is zero or more of the following values added together.

Value	Description
0	Normal file.
1	Read-only file.
2	Hidden file.
4	System file.
8	Volume label.
16	MS-DOS directory.
32	File has changes since last backup.

big-endian Multiple byte data values (not strings) are stored with the highest order byte first. For example, the long integer

&H01020304 is stored as this sequence of four bytes: &H01, &H02, &H03 and &H04. A Binary or Random file written using Put uses little-endian format so that it can be read using Get on any machine. (Big-endian machines, like the Power-PC, reverse the bytes as they are read by Get or written by Put.)

See Also: Dir( ), GetAttr( ), SetAttr( ).

charlist A group of one or more characters enclosed by [ ] as part of Like operator's right string expression.

- This list contains single characters and/or character ranges which describe the characters in the list.
- A range of characters is indicated with a hyphen (-) between two characters. The first character must be ordinally less than or equal to the second character.
- Special pattern characters like ?, *, # and [can be matched as literal characters.
- The] character can not be part of charlist, but it can be part of the pattern outside the charlist.

condexpr An expression that returns a numeric result. If the result is zero then the conditional is False. If the result is non-zero then the conditional is True.

```
0 false
-1 true
X > 20 true if X is greater than 20
S$ = hello true if S$ equals hello
```

dateexpr An expression that returns a *date* result. Use #literal-date# to express a date value.

```
#1/1/2000#  Jan 1, 2000
Now+7  seven days from now
DateSerial(Year(Now)+1,Month(Now),Day(Now))
one year from now
```

dialogfunc A dialog function executes while a *UserDialog* is visible.

dim [lower To] upper

Array dimension. If lower is omitted then the lower bound is zero. upper must be at least as big as lower.

Dim A(100 To 200) '101 values

Note: For ReDim the lower and upper may be any valid expression. Otherwise, lower and upper must be constant expressions.

dlgvar A dialog variable holds values for fields in the dialog. Dialog variables are declared using **Dim** dlgvar As *UserDialog*.

expr An expression that returns the appropriate result.

field Use .field to access individual fields in a dialog variable.

```
dlg.Name$
dlg.ZipCode
```

instruction A single command.

```
Beep
Debug.Print Hello
Today = Date
```

Multiple instructions may be used instead of a single instruction by separating the single instructions with colons.

```
X = 1:Debug.Print X
If X = 1 Then Debug.Print X=;X:Stop
Beep  must resume from Stop to get to here
```

label An identifier that *names* a statement. Identifiers start with a letter. Following chars may be a letter, an underscore or a digit.

little-endian Multiple byte data values (not strings) are stored with the lowest order byte first. For example, the long integer &H01020304 is stored as this sequence of four bytes: &H04, &H03, &H02 and &H01. A Binary or Random file written using Put uses little-endian format so that it can be read using Get on any machine. (Big-endian machines, like the Power-PC, reverse the bytes as they are read by Get or written by Put.)

macro A macro is like an application. Execution starts at the macro's Sub Main.

method An object provides methods and *properties*. Methods can be called as subs (the return value is ignored), or used as functions (the return value is used).

If the method name contains characters that are not legal in a *name*, surround the method name with [].

```
App.[Title$]
```

module A file with public symbols that are accessible by other modules/macros via the #Uses comment.

- A module is loaded on demand.
- A code module is a code library.

- An object module or class module implements an OLE automation object.
- A module may also access other modules with its own #Uses comments.

name An identifier that names a variable or a user defined subroutine, function or property. Identifiers start with a letter. Following chars may be a letter, an underscore or a digit.

```
Count
DaysTill2000
Get_Data
```

num An expression that returns a numeric result. Use &O to express an octal number. Use &H to express a hex number.

```
10236
3.14159
1.2E12
Count
Count-1
InStr(S$,"A")
&O100 64
&H100 256
```

numvar A variable that holds one numeric value. The name of a numeric variable may be followed by the appropriate *type* char.

objexpr A expression that returns a reference to an object.

```
CreateObject(WinWrap.CDemoApplication)
```

objtype A specific OLE type defined by your application, another application or by an object module or class module.

See Also: Objects, CreateObject( ), GetObject( ).

objvar A variable that holds a *objexpr* which references an object. Object variables are declared using As *Object* in a **Dim**, **Private** or **Public** statement.

param [[Optional] [| ByVal | ByRef] | ParamArray] param[type][( )] [As type]

The *param* receives the value of the associated expression in the subroutine, function or property call. (See *arglist*.)

An Optional *param* may be omitted from the call. It must be a **Variant** type. All parameters following an Optional parameter must also be Optional.

ParamArray may be used on the final *param*. It must be an array of **Variant** type. It must not follow any Optional parameters. The ParamArray receives all the expressions at the end of the call as an array. If **LBound**(*param*) **UBound**(*param*) then the ParamArray didn't receive any expressions.

If the *param* is not ByVal and the expression is merely a variable then the *param* is a reference to that variable (ByRef). (Changing *param* changes the variable.) Otherwise, the parameter variable is local to the subroutine, function or property, so changing its value does not affect the caller.

Use *param*( ) to specify an array parameter. An array parameter must be referenced and can not be passed by value. The bounds of the parameter array are available via **LBound( )** and **UBound( )**.

Property Get, Let and Set blocks do not allow Optional or ParamArray parameter types.

precedence

When several operators are used in an expression, each operator is evaluated in a predetermined order. Operators are evaluated in this order:

^(power)

- (negate)

* (multiply), / (divide)

\ (integer divide)

Mod (integer remainder)

+ (add), - (difference)

& (string concatenate)

= (equal), <> (not equal), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), **Is** (object equivalence)

Not (logical bitwise invert)

And (logical bitwise and)

Or (logical or bitwise or)

Xor (logical or bitwise exclusive-or)

Eqv (logical or bitwise equivalence)

Imp (logical or bitwise implication)

Operators shown on the same line are evaluated from left to right.

property An object provides *methods* and properties. Properties may be used as values (like a function call) or changed (using assignment syntax).

If the property name contains characters that are not legal in a *name*, surround the property name with [].

```
App.[Title$]
```

statement One or more *instructions*. A statement is at least one macro line long. **Begin** Dialog, **Do**, **For**, **If** (multiline), **Select Case**, **While** and **With** statements are always more than one line long. A single line statement continues on the next line if it ends a line with a space and an underscore _.

```
S$ = This long string is easier to read,  +
_
if it is broken across two lines.
Debug.Print S$
```

str An expression that returns a string result.

```
Hello
S$
S$ +  GoodbyeS$ &  Goodbye
Mid$(S$,2)
```

strarray A variable that holds an array of string values. The name of a string variable may be followed by a $.

strvar A variable that holds one string value. The name of a string variable may be followed by a $.

```
FirstName$
```

type Variable and argument types, as well as, function and property results may be specified using a type character as the last character in their name.

Type char	As Type
%	Integer
&	Long
!	Single
#	Double
@@	Currency
$	String

userenum User defined enums are defined with Enum.

usertype User-defined types are defined with **Type**.

usertypevar	A user-defined type variable holds values for elements of the user-defined type. Use r-defined types are defined using **Type**. User-defined variables are declared using **Dim**, **Private** or **Public**.
var	A variable holds either a string, a numeric value or an array of values depending on its type.
variantvar	A variant variable holds any type of value (except *String*n* or *usertypevar*).

User Notes

Appendix B

Error Codes

The following table lists all error codes with the associated error text.

Error #	Description
10000	Macro execution interrupted.
10001	Out of memory.
10008	Invalid '#Uses "module" comment.
10009	Invalid '#Uses module dependency.
10010	Macro is already running.
10011	Cant allocate memory to macro.
10012	Macro has syntax errors.
10013	Macro does not exist.
10014	Another macro is paused and cant continue at this time.
10017	No macro is currently active.
10018	Subroutine does not exist.
10019	Wrong number of parameters.
10021	Cant allocate large array.
10022	Array is not dimensioned.
10023	Array index out of range.
10024	Array lower bound is larger than upper bound.
10025	Array has a different number of indexes.
10030	User dialog has not been defined.
10031	User pressed cancel.
10032	User dialog item id is out of range.
10033	No UserDialog is currently displayed.
10034	Current UserDialog is inaccessible.
10035	Wrong with, don't GOTO into or out of With blocks.
10040	Module could not be loaded.
10041	Function not found in module.
10048	File not opened with read access.
10049	File not opened with write access.
10050	Record length exceeded.
10051	Could not open file.
10052	File is not open.
10053	Attempt to read past end-of-file.

10054	Expecting a stream number 1, 2, 3 or 4.
10055	Input does not match var type.
10056	Expecting a length in the range 1 to 32767.
10057	Stream number is already open.
10058	File opened in the wrong mode for this operation.
10059	Error occurred during file operation.
10060	Expression has an invalid floating point operation.
10061	Divide by zero.
10062	Overflow.
10063	Expression under flowed minimum representation.
10064	Expression loss of precision in representation.
10069	String value is not a valid number.
10071	Resume can only be used in an On Error handler.
10075	Null value cant be used here.
10080	Type mismatch.
10081	Type mismatch for parameter #1.
10082	Type mismatch for parameter #2.
10083	Type mismatch for parameter #3.
10084	Type mismatch for parameter #4.
10085	Type mismatch for parameter #5.
10086	Type mismatch for parameter #6.
10087	Type mismatch for parameter #7.
10088	Type mismatch for parameter #8.
10089	Type mismatch for parameter #9.
10090	OLE Automation error.
10091	OLE Automation: no such property or method.
10092	OLE Automation: server cannot create object.
10093	OLE Automation: server cannot load file.
10094	OLE Automation: Object var is Nothing.
10095	OLE Automation: server could not be found.
10096	OLE Automation: no object currently active.
10097	OLE Automation: wrong number of parameters.
10098	OLE Automation: bad index.
10099	OLE Automation: no such named parameter.
10100	Directory could not be found.
10101	File could not be killed.
10102	Directory could not be created.
10103	File could not be renamed.
10104	Directory could not be removed.

10105	Drive not found.
10106	Source file could not be opened.
10107	Destination file could not be created.
10108	Source file could not be completely read.
10109	Destination file could not be completely written.
10110	Missing close brace }.
10111	Invalid key name.
10112	Missing close paren).
10113	Missing close bracket].
10114	Missing comma ,.
10115	Missing semi-colon ;.
10116	SendKeys couldn't install the Windows journal playback hook.
10119	String too long (too many keys).
10120	Window could not be found.
10130	DDE is not available.
10131	Too many simultaneous DDE conversations.
10132	Invalid channel number.
10133	DDE operation did not complete in time.
10134	DDE server died.
10135	DDE operation failed.
10140	Cant access the clipboard.
10150	Window style must be in the range from 1 to 9.
10151	Shell failed.
10160	Declare is not implemented.
10200	Basic is halted due to an unrecoverable error condition.
10201	Basic is busy and can't provide the requested service.
10202	Basic call failed.
10203	Handler property: prototype specification is invalid.
10204	Handler is already in use.

User Notes

Index

O

P

Q

R

S

T

V

W