Contents

The following Help Topics are available:

The Editor

The QmodemPro for Windows 95 editor can be used to edit scripts or text files. If you have used other word processors and text editors in Windows, you should be familiar with most commands, icons and shortcut keys used by the *QmodemPro for Windows 95* Editor.

The Script Editor

If you choose the **Scripts/Edit** command from the *QmodemPro for Windows 95* main menu to open the Qmodem Editor, the Editor will open with special features used especially for creating and editing scripts.

You can also launch the script editor by using theStart/Run command from the taskbar and typing:

```
C:\QMWIN\QMEDITOR.EXE /SCRIPT
```

The primary differences between the two versions of the Editor deal with scripting manipulation. The Script Editor has a built-in Compiler feature, to allow quick compiling while you are editing.

It also offers a Syntax Highlight option. This option allows you to show specific types of script language (syntax) in user-defined colors. This allows you to visually differentiate syntax at a glance. For a detailed list of these features, refer to Book 2 of the QmodemPro for Windows 95 documentation, the SLIQ guide.

Keyboard Shortcuts

CTRL+O	Open an existing file
CTRL+S	Save the current file
CTRL+P	Print the current file
CTRL+N	Create new file
CTRL+T	Delete word right
CTRL+V	Paste from Windows clipboard
CTRL+X	Cut marked text to Windows clipboard
CTRL+Y	Delete line
CTRL+C	Cut to Windows clipboard
CTRL+H	Paste from Windows clipboard
CTRL+Z	Undo editing
CTRL+F	Find text
CTRL+HOME	Top of screen
CTRL+END	End of file

Launching the Editor

You can launch the Qmodem Editor from a *QmodemPro for Windows 95* toolbar icon, if you have configured the toolbar to include this icon. To add the Editor icon, click **Tools/Customize** and select Toolbar, or click the right mouse button on the toolbar and select Properties. Highlight the Editor option in the Available Buttons section, and double click or click Add to add it to Toolbar Buttons. The icon will be added to the toolbar.

You can also open the Editor from a command line without even opening *QmodemPro for Windows 95* . From a command line (choose Start/Run from the Windows Taskbar), type

```
C:\QMWIN\QMEDITOR.EXE
```

or

```
C:\QMWIN\QMEDITOR.EXE /SCRIPT
```

to launch the Script Editor. If you have installed *QmodemPro for Windows 95* on a different drive, change the command line to reflect the appropriate drive and directory for your particular setup.

Marking Text

You can mark text with the mouse. After the text is marked, press CTRL and hold down the left mouse button. Move the cursor to the place where you want to copy to, and release the mouse button.

Another alternative is to mark the text with the mouse and then click the right mouse button on the highlighted text.

Moving Text

First, mark the text you want to move. Then cut the marked text to the Windows Clipboard using your mouse to select the menu command **Edit/Cut**. Or press SHIFT DEL.

Then move the insertion point to the place where you would like to copy the selected text. Using your mouse, select the menu command **Edit/Paste**. Or, press CTRL+V.

Another alternative is to mark the text with the mouse and then drag the text to the new position.

Copying Text

First, mark the text you want to copy. Then copy the marked text to the Windows Clipboard using your mouse to select the menu command **Edit/Copy**. Or, press CTRL+C.

Then move the insertion point to the place where you would like to copy the selected text. Using your mouse, select the menu command **Edit/Paste**. Or, press CTRL+V.

Another alternative is to mark the text with the mouse and then click the right mouse button on the highlighted text. Select **Copy** from the drop down box options.

Deleting Text

Use the backspace or DEL key when you want to delete text from the current document but you have text on the Windows Clipboard that you want to keep.

If you want to cut text but save it to the Windows Clipboard, highlight the text you want to cut, then use the Cut icon (it looks like little scissors) or CTRL+X

Another way to delete text is to mark the text with the mouse and then click the right mouse button on the highlighted text. Select **Delete** from the drop down box options.

To delete one character at a time, press backspace to delete the character to the left of the insertion point. Press DEL to delete the character to the right of the insertion point.

To delete more than one character, select the text you want to delete. Press backspace or DEL. To place the text onto the Windows Clipboard, choose **Cut** from the **Edit** menu.

Undo/Redo

You can **redo** changes, and restore your text to the way it was before you undid the last command, with the **Edit/Undo** command or CTRL+A.

You can **undo** changes, and restore your text to the way it was before you performed the most recent command, with the **Edit/Undo** command or CTRL+Z.

Finding Text

You can start a search for specific text at any point in a document. To find specific characters or words, move the insertion point to the place you want the search to begin. From the Edit menu, choose **Find**. Type in the text you want to find.

If you want to match case exactly, select the **Case Sensitive** option.

To specify the search direction, select the **Up or Down** option. Then choose the **Find Next** button.

To find the next occurrence of the text, choose the **Find Next** button again.

You can also use the **Find** icon on the toolbar to search for text. Click the icon (it looks like a magnifying glass) and a **Find** dialog box will appear.

You can replace specific text by using the **Replace** command. Much the same way that **Find** and **Find Next** find text, **Replace** will allow you to replace specific text. Enter the text you want to get rid of, and the text you want to replace it with. If you want to match capitalization exactly, select the **Match Case** check box. Click on **Find Next**, verify that you want the text replaced, and click **Replace.**

You can use the **Edit/Replace All** command to globally replace text in your file.

Print/Send

You can print your file by selecting **File/Print**, or pressing CTRL+P. An options box will appear, allowing you to select the printer to use, page range, the number of copies, and print setup.

The Qmodem Editor has a built in Print Preview feature. Before printing a text file, you can click on File/Print Preview to see a single or multiple page preview of how the document will look before you print it.

Files open in the Editor can be sent directly as a mail message, appended to a mail message, or as a fax directly from the Editor. Selecting the **File/Send** menu command from the Editor main menu will launch Microsoft Exchange, allowing your file to be included as all or part of any Exchange application.

Changing the Look of the Editor

Use the Editor's View command to change the look of the Editor. You can turn off the toolbar or status bar with the **View/Toolbar** or **View Status bar** command. Choose Toolbar to toggle the toolbar on and off. If the toolbar is visible, a checkmark will appear beside it when you use this command. Toggling the status bar on and off is done in exactly the same way as the toolbar, using the **View/Status bar** command.

The View/Options/Fonts command will bring up a property sheet with options for setting the default fonts for the edit window. **View/Options/Editor** lets you set tab definitions, word wrap, and auto indent options.

The View Menu

The View menu allows you to change the look of the Editor.

Toolbar Toggles the Toolbar Visible command on and off. When **on**, the toolbar is visible from the editor window.

Statusbar Toggles the Statusbar Visible command on and off. When **on**, the statusbar is visible from the editor window.

Options Allows you to set fonts, tabs, word wrap, and auto-indent options.

The Edit Menu

The *QmodemPro for Windows 95* edit menu allows you to manipulate data inside files, with common editing commands. Here is a list of options, and a brief description of their function available in the Edit menu.

Undo   Undoes (reverses) the last action you performed.

Redo   Restores the document to the way it was before you performed an undo.“Undoes” the last “Undo”.

Cut   Deletes the marked selection from your document and places it into the Winodws clipboard.

Copy   Copies the marked selection of your document to the Windows clipboard without deleting it from you document.

Paste   Pastes the contents of the Windows Clipboard to the current cursor position.

Delete   Deletes the marked selection from your document.

Find   Opens a Find dialog box, allowing you to specify text to find in the currently active document.

Replace   Works in conjunction with the Find command. Allows you to globally search and replace text within your document.

Goto   Places the cursor at a specified line number in the currently active document.

Find Error (Script Editor Only)   Finds and moves the cursor to a run time error in the source code.

The File Menu

The *QmodemPro for Windows 95* Editor file menu has options allowing you to open, save, and print files. Here is a list of options, and a brief description of their function in the File menu.

New Opens a new file. The CTRL+N keys can also be used to open a new file.

Open Opens the Windows common dialog box, allowing you to choose the file you want to open from available drives and directories. The CTRL+O keys will also open this dialog box.

Save Saves the currently active file, including any changes you have made. Overwrites the original file. You can also use the key command CTRL+S to save a file. If you want to save changes without overwriting the original, use Save As. Because of Microsoft's Win95 requirements, the Qmodem Editor file extension is now '.QED' (Qmodem Editor File) instead of '.TXT'.

Save As Opens a Save As dialog box, allowing you to save changes to your document by saving it with another filename. This allows you to keep your original document intact.

Print Preview Shows a representation of the entire document as it will look when printed.

Print Setup Opens the Windows Print Setup dialog box, allowing you to change the setup of your printer, including which printer to use, the printer properties, paper orientation, and paper source.

Send Allows you to send the document directly to an E-mail message. Opens Microsoft Exchange or your specified electronic messaging system.

The Compile Menu

The Script Editor offers an extra command menu, the Compile menu. This offers features specific to editing scripts. The features on the Compile menu are:

Compile Compiles source code active in the Editor window.

Primary File Defines the focus file for the compile. This defines the primary file when more than one file is being used. When a primary file has been defined, the filename will appear next to this command in the menu.

Clear Primary File Clears the primary file, allowing you to define a new file as primary.

Scripts and SLIQ

What is a script? A script program is a set of instructions that you give *QmodemPro for Windows 95* that tells it how to perform a particular task. Its structure and syntax is similar in many ways to the popular Basic programming language, but you don't need to be a programmer to make use of Script Language Interface for QmodemPro for Windows 95 (SLIQ).

Depending on the complexity of its construction, a SLIQ script can be thought of as a command shortcut, a macro or a computer program. Scripts are an ideal way to automate repetitive tasks, but can also be used to create custom options for the flow of your telecommunications activity. A SLIQ script can make use of *QmodemPro for Windows 95* program functions as well as internal functions within Windows 95 itself.

SLIQ scripts are composed of lines of text that represent the activities to be performed. It's a compiled script language, meaning that the text representation of the script is converted to a binary file when it is complete. Compiling scripts not only allows them to execute faster, but lets you distribute executable copies of your scripts without the underlying text source code.

For the large majority of *QmodemPro for Windows 95* users, script usage will be limited to grouping a sequence of tasks into a single command, usually by using the QuickLearn feature. QuickLearn provides an interface to SLIQ that is as easy to use as a tape recorder; you just turn it on and it records your task for playback at a later time.

What can I do with a script?

SLIQ provides the means of performing almost any telecommunications activity desired.

To learn more about scripts and SLIQ, select a topic below. Refer to the SLIQ Script Guide (volume 2 in your documentation) for a more complete explanation of the SLIQ language.

Design Elements of a Script

QuickLearn

Compiling a Script

Debugging a Script

Running a Script

Stopping a Script

General Information

What can I do with a script?

Automated logon

While the steps to connect to various on-line services, host computers and BBS systems are very straightforward in *QmodemPro for Windows 95*, each host has its own procedures for getting your name, password and other information. Automation of this process is probably the most common application for a script program. It is also the one that lends itself best to the QuickLearn feature since it is repetitive in nature.

File transfers

Automating the process of downloading files from a remote system is another application where scripts are handy. Although the QuickLearn feature can also be used for this type of activity, it may require modification to control the desired download directories or other download parameters.

Fully automated sessions

Combining the two previous examples you can see that a more complex script is capable of logging on to a remote system, downloading files and perhaps additional activity such as retrieving new messages. Any activity that can be performed manually with *QmodemPro for Windows 95* can be automated using SLIQ.

Unattended QmodemPro for Windows 95 operations

Perhaps the best example of an unattended script is the *QmodemPro for Windows 95* host mode. The functionality of host mode is discussed in the User Guide, but the actual operation of host mode is driven by a series of pre-written scripts that are included with *QmodemPro for Windows 95*. These scripts and several others will be used as examples as we explore the power of SLIQ.

Design elements

SLIQ is patterned after the BASIC programming language. This decision is based on the desire to incorporate a number of advanced programming structures and concepts, while appealing to the largest audience. If you are familiar with BASIC you will be accustomed to many aspects of SLIQ. There will be some new commands specific to *QmodemPro for Windows 95* but you should feel very comfortable with writing and editing scripts.

SLIQ scripts must be compiled to an executable format before they can be used. Compiled scripts are advantageous because they execute faster than interpreted scripts and allow the author to keep the script source code confidential. If an attempt is made to execute a script before it has been compiled, the compiler is automatically invoked by the program. Once compiled, either manually or automatically, a script is executed immediately when requested.

Scripts are stored in the directory specified for script files which is selected with the menu choice **Tools/Options/Paths**. SLIQ files use standard Windows 95 filenames with two different extensions. The source script files created with your editor uses the extension **.QSC** and when they are compiled into an executable form by SLIQ they are given the extension **.QSX**.

Script files are created and modified using any standard ASCII text editor. *QmodemPro for Windows 95* includes an internal editor that you access from the menu choices **Scripts/Edit** or **Tools/Editor**. The internal editor supports auto-indent and tabs for structuring your script source files in a readable format, however you are free to use an external editor or programming environment if desired.

How are scripts created?

The QuickLearn feature

The *QmodemPro for Windows 95* script language can be used for many applications without the need to learn anything about scripting or SLIQ. QuickLearn lets you to record keystrokes for playback at a later time and is a simple process.

The first way of starting a QuickLearn script is to revise a Phonebook entry and place the name of a new script in the Scripts field. The next time you dial this entry, *QmodemPro for Windows 95* will automatically record a script file.

The second way to create a QuickLearn script is to use the pull down menu choice **Scripts/QuickLearn** to begin recording a script.

When activated, *QmodemPro for Windows 95* displays the standard file selection dialog and request a name for the script to be created. All scripts should be saved in the directory designated for scripts which by default is C:\QMWIN\SCRIPTS. If your configuration is different you should save your scripts in the correct script directory.

Any valid Windows 95 filename may be used and the extension must be **.QSC**. This extension is automatically added when script names are specified in the script filename dialog box.

Once a filename is selected you are returned to terminal mode and every keystroke is recorded until QuickLearn mode is terminated by again selecting the **Scripts/QuickLearn** menu choice. While recording is active the terminal window status line displays the name of the script preceded by the “greater than” symbol (>) indicating that recording is active. In addition, a check mark is placed in front of the QuickLearn menu option.

Scripts created using QuickLearn can be viewed and edited just as any other script. The menu choice **Scripts/Edit** can be used to take a look at the scripts created with SLIQ’s QuickLearn feature.

While QuickLearn is an extremely convenient facility, it has its limitations. QuickLearn can record your interactions with a remote computer and replicate them exactly, but it cannot add decision-making logic to the scripts it creates, or account for unexpected or changing conditions. When you run out of QuickLearn power, it’s time to write or edit your own scripts.

Script programming

Writing a new script from the ground up is a necessity in some cases. Any standard ASCII text editor can be used to create or modify a SLIQ script, including the *QmodemPro for Windows 95* internal editor.

To create a new script select the menu choice **Scripts/Edit** to display the standard file selection dialog box. The default directory in the dialog will display files in your scripts directory as defined in **Tools/Options/Paths**, and the default extension will be **.QSC**. Create a new script file by entering the name of the script to be created. It is not necessary to add the script extension **.QSC** since it is added automatically when the **Scripts/Edit** menu command is used to create a script file. Note that the alternate method of invoking the editor, using the menu choice **Tools/Editor/Edit a File** does *not* automatically add the extension **.QSC**.

To edit a SLIQ script use the same procedure as creating a new script, but select the desired script from the dialog box.

Compiling a script

SLIQ is a compiled language. This means that the text of every **.QSC** text file is converted to executable code before being used by *QmodemPro for Windows 95*. Script source files (those ending in **.QSC**) are compiled either automatically or manually.

After creating a SLIQ script file you can manually perform a compile by selecting the **Scripts/Compile** menu choice. The compile process will read your script, check each line for proper syntax and create an executable **.QSX** file with the same name as the source file. If any errors are encountered the compile process will stop with a compile error message and invoke the editor. The editor cursor will be located as near the error as possible, enabling you to make the necessary changes. A complete list of compiler error messages can be found in the sections **Compiler Errors and Script Debugging**, later in this manual.

If a script has been created or modified but not yet compiled when a request to execute is received, the compile process is automatically invoked. The compile process may take a few seconds to several minutes depending on the size and complexity of the script.

Debugging a script

Regardless of the simplicity of a script or the capabilities of the programmer, scripts are seldom written perfectly on the first attempt. Many times an error is simply a typographical mistake or improper syntax for a command. In these cases the errors are caught during the compile cycle and the editor cursor is placed at the approximate location of the error in the script source. This type of error is usually relatively easy to correct.

The harder type of error to locate and correct involves a script that follows correct syntax and compile guidelines, but fails to produce the desired results. In this case the script debugger is an invaluable tool. The debugger allows you to execute a script line-by-line while viewing the results. It can skip sections of code and even display the changing value of variables as each line is executed.

The operation of the debugger is discussed in its own section after the SLIQ language documentation. See the **Compiler Errors** section under **Debugging** for details.

Running a script

Scripts can be executed in several different ways:

1. Scripts can be started with the menu command **Scripts/Execute**. When this option is used, you can select either the **.QSC** or **.QSX** filename for the desired script if both exist. If only the **.QSC** source filename exists, the script will be compiled to a **.QSX** file before execution.
2. Scripts can be executed by clicking on the toolbar button labeled **Execute Script**. This action displays the same dialog box that is called when the **Scripts/Execute** command is selected.
3. Scripts can be started automatically every time a Phonebook entry is dialed. A script name is associated with a specific dialing directory entry in the *Script* field for entries in the *QmodemPro for Windows 95* Phonebook. When connection is made, the script is executed automatically. These types of scripts are often referred to as *linked scripts* because they're linked to phonebook entries. The same script can be tied to multiple Phonebook entries, and can be used to take control immediately after a connection to automate repetitive logon processes. Linked scripts that automate logon activities are also called *logon scripts*.
4. Scripts can be started by clicking a macro key on the macro bar at the bottom of the screen that has been assigned to a script name using the @SCRIPT macro command. Once started, macro-linked scripts operate the same as a script executed from the pull-down menu.
5. Scripts can be executed immediately when *QmodemPro for Windows 95* is started. The command line syntax for launching *QmodemPro for Windows 95* with a script is:

1. `QMWIN FILENAME.QSX`

1. or

1. `QMWIN FILENAME.QSC`

1. which will cause a compile if needed.
2. Once initialization is complete, *QmodemPro for Windows 95* will load and execute the script specified on the command line.

Script extensions .QSX and .QSC were automatically registered with Windows 95 when *QmodemPro for Windows 95* was installed. This association allows you to double-click on the script file name within Explorer to launch *QmodemPro for Windows 95* with the selected script as a command line argument.

Regardless of how a script is invoked, it operates in the same manner. The Script icon on the toolbar is displayed in a depressed position during script execution and the terminal windows status line displays the name of the script being run.

Stopping a script

You can stop a script at any time in several ways. The easiest is to click on the **Script** icon on the toolbar. A confirmation prompt dialog box is displayed before execution is terminated.

You can also toggle the menu selection **Scripts/Execute** to stop a script. This menu item displays a check mark during execution, which is removed after confirming that you want to stop the script.(New topic text goes here.)

General Program Information

Every variation of BASIC is slightly different, and SLIQ is no exception. This section provides information about the various items that make up SLIQ and how they operate. It only deals with the underlying operational characteristics of the language, not the command set. The entire set of SLIQ commands are listed with examples in the SLIQ Script Guide.
See also Constants, Types, Variables, Expressions, Statements, Multiple line IF statements, Loop statements, and Subroutines and Functions.

Tokens

Tokens are the smallest meaningful units of text in a *QmodemPro for Windows 95* script program. There are five kinds of tokens: special symbols, numbers, keywords, identifiers, and string constants.

Special Symbols

Besides letters, numbers, and spaces, SLIQ accepts the following special characters and pairs of characters:

```
+ - * / = < > . , ( ) : ; ' " <= >= <>
```

Numbers

SLIQ accepts numbers in ordinary decimal format. Hexadecimal numbers are represented using 0x or &H as a prefix to the number. Engineering notation (e or E, followed by an exponent) is read as “times ten to the power of” in real numbers. For example, 7E-2 means 7 x 10-2.

Keywords

Keywords appear in lower case typewriter font throughout this section. The following is a complete list of SLIQ keywords:

```
access alias all and append as binary byval call caption case catch close
const declare dial dialog dialogbox dim div do else elseif end eqv exit flush
font for function get gosub goto if imp include input is len let lib lock
loop mid mod name next not open or output print put random read receive rem
return seek select send shared static step sub then to type unlock until wend
when while write xor
```

These keywords cannot be used as identifiers. Since SLIQ is not case sensitive, these keywords may appear in any combination of upper or lower case in your scripts.

Identifiers

Identifiers denote constants, types, variables, subroutines, functions, and fields in user defined types. An identifier can be of any length, but shorter identifiers are usually easier to manage. An identifier must begin with a letter and can not contain spaces. Letters, digits, and underscore characters are allowed after the first character. Like keywords, identifiers are not case sensitive.

String Constants

A character string is a sequence of zero or more characters from the extended ASCII character set, written on one line in the program and enclosed by double quote characters ("). Case is significant in string constants; "Hello" is a different string from "HELLO".

Comments

Comments consist of text the compiler should ignore, but are useful for annotating your SLIQ programs. Comments are preceded by either the rem statement, a double forward slash (//) or the ‘ character (single apostrophe).

Note that rem is treated as a regular statement, so if you want to place a comment at the end of a line using rem , you must precede the rem by a colon to separate it from the previous statement. The single apostrophe can be used without the need for a separating colon:

```
print "Hello world!"    'comment OK

print "Hello world!"    :rem comment OK
```

Note that the first needs no colon while the second does.

Constants

A constant is an identifier that represents a value that can’t change. A constant is declared in a SLIQ program like this:

```
const Tries = 10
```

This declaration causes the value 10 to be substituted wherever the identifier Tries appears in the program. You can also define constants in terms of other constants using expressions, like this:

```
const LookingBad = Tries - 3
```

This will cause the value 7 to be used wherever LookingBad appears in the program.

Types

SLIQ is a typed language, which means that each variable or expression has an associated type. This type defines the kind and size of values that the variable can hold, as well as the operations that can be performed on that variable. The following identifiers represent the predefined types in the language:

```
byte
short
integer (or long)
real
string
```

Simple types

There are five simple types whose characteristics are shown below:

Type	Range	Format
byte	0 to 255	8 bit unsigned
short	-32768 to 32767	16 bit signed
integer (long)	-2147483648 to 214783647	32 bit signed
real	1.5e-45 to 3.4e38	32 bit IEEE
string	0 to 32767 characters in length	variable length

Strings can also be declared to be of fixed length, which means that they always hold a specific number of characters (even if those characters are just spaces). Here is an example of a declaration of a string of length 30:

```
dim name as string*30
```

In this case name will always hold exactly 30 characters. Fixed length strings are particularly useful in user defined types (see below) because variable length strings are not permitted in a user defined type declaration.

Arrays

Arrays are sequences of values, all of which have the same type. For example, the following code declares an array of 10 integers:

```
dim a(10) as integer
```

Actually, there are 11 integers declared here; they are referenced as a(0), a(1), through a(10). The type of elements in an array can be of any type except another array.

User Defined Types

A user defined type is a type that holds more than one value. Unlike an array, the values that a user defined type holds can be of different types. For example, here is a user defined type that can be used to hold a date value:

```
type date
  year as integer
  month as integer
  day as integer
end type
```

In this example all the fields are of the same type. Here is another example that could be used to hold a person’s information:

```
type person
  name as string*50
  phone as string*20
  birthdate as date
end type
```

Note that the date type declared above can be used inside another type declaration. User defined types can be nested in this way as many levels deep as necessary.

Dialog Box Types

SLIQ supports Windows 95 dialog boxes using a mechanism similar to user defined types. For information on declaring and using dialog boxes, see the separate section on Dialog Boxes at the end of chapter 2.

Variables

A variable is an identifier that represents a value that can change. A variable is declared in a SLIQ program like this:

```
dim count as integer
```

The word “count” can then be used in any expression where an integer is accepted. To declare an array of values, place the number of values you want to declare in parentheses after the variable identifier. For example,

```
dim friends(10) as string
```

This causes friends(0) through friends(10) to be declared. Declaring a user defined type works the same way:

```
dim bob as person

bob.name = "Bob Smith"

bob.phone = "555-1212"

bob.birthdate.year = 1963

bob.birthdate.month = 4

bob.birthdate.day = 17
```

The above statements also assign values to each of bob’s fields. As you can see, we have built upon our previous examples to create a personal record.

Expressions

Expressions are made up of operators and operands. Most operators are binary and take two operands. Two operators are unary and only take one operand. Binary operators use the usual algebraic form (for example, A + B). A unary operator always precedes its operand (for example, -B). There are two unary operators, not and -. Not is a bitwise logical not of its operand, and - takes the negative of its operand. The following is a list of the binary operators in SLIQ, in order of precedence from highest to lowest:

Operator	Notes
^	exponentiation
* / \ mod	multiplication, division, integer division or same as, modulus
+ -	addition, subtraction
= <> < <= > >=	equality and inequality operators
and	bitwise logical and
or	bitwise logical or
xor eqv	exclusive or, equivalence ((x eqv y) = not (x xor y))
imp	implication, ((x imp y) = ((not a) or b))

Parentheses are used to modify the above precedence order. Expressions within parentheses are always evaluated starting with the innermost set of parentheses. For example,

Expression	Notes
3 + 4 * 5 = 23	(multiplication takes precedence over addition)
(3 + 4) * 5 = 35	

Functions are called by naming the function to call, perhaps followed by a parameter list in parentheses after the function name. For example, if s is a string,

```
len(s)
```

is an integer type expression equal to the length of the string s.

Statements

Statements are commands that tell SLIQ what to do. Statements can be broken into two major classes — those that control the execution of other statements and those that don’t.

Simple statements stand by themselves and don’t control the execution of other statements. A simple statement is our first example:

```
print "Hello world!"
```

Compound statements directly control the execution of other statements. Each type of compound statement is described in the following sections:

Single line if statement

The simplest sort of compound statement is the single-line if statement:

```
if x = 5 then print "x is five"
```

This causes the print statement to be executed only if x currently has the value 5. If x does not have the value 5, then the print statement is skipped.

Multiple line if statement

If statements can also span multiple lines, as in this example:

```
if x = 5 then
  print "x is five"
  x = x + 1
  print "x is now six"
end if
```

The style of indenting the lines that are controlled by the if statement is not required, but it makes the program easier to read. You will see this style in each of the following compound statement examples. If statements can also have an else directive:

```
if x >= 1 and x <= 10 then
  print "x is between 1 and 10"
else
  print "x is less than 1 or greater than 10"
end if
```

Finally, if statements can have an “elseif” clause, which helps make certain constructs easier to write (see the select case statement below for another approach):

```
if x >= 1 and x <= 10 then
  print "x is between 1 and 10"
elseif x > 10 then
  print "x is greater than 10"
else
  print "x is less than 1"
end if
```

Loop statements

The loop compound statement comes in four flavors which differ in when and how the condition to exit the loop is tested. In general, the loop statement will repetitively execute a group of statements until a particular condition is met. For example,

```
x = 1
do while x <= 5
  print "x is "; x
  x = x + 1
loop
```

This is what is called a top-tested loop — the condition is tested at the top of the loop before any of the statements inside the loop are executed. In this case the loop will be executed five times — once for each value of x from 1 to 5. The other kind of top-tested loop is shown below:

```
x = 1
do until x > 5
  print "x is "; x
  x = x + 1
loop
```

The above example functions exactly the same as the first, the difference is the kind of test that is performed to see whether the loop should exit.

Bottom-tested loops work in a similar way, except that the test to see whether the loop should exit occurs after all the statements in the loop. The statements in a bottom-tested loop are always executed at least once (in a top-tested loop the statements may not be executed at all).

```
x = 1
do
  print "x is "; x
  x = x + 1
loop while x <= 5
```

or

```
x = 1
do
  print "x is "; x
  x = x + 1
loop until x > 5
```

An alternate form of the top-tested while loop is shown below:

```
x = 1
while x <= 5
  print "x is "; x
  x = x + 1
wend
```

Select case statement

The select case statement is a convenient way of testing a number of similar conditions. For example, the following statement tests the value of a variable x:

```
select case x
  case 1
    print "x is 1"
  case 2 to 5
    print "x is between 2 and 5"
  case is > 10
    print "x is greater than 10"
  case else
    print "x is less than 1 or between 6 and 10"
end case
```

There are several different types of cases that can be tested: single value, range of values, and a relation. An example of each is shown in the above code example. A single case value matches if the case variable is exactly the same as the value given. A range case checks the case variable against the given values to see whether it is within the given range including the endpoints. In the above example, x would be tested as “x >= 2 and x <= 5”. A relation case is denoted by the keyword is, followed by a binary operator, followed by a value. The keyword is is replaced by the case variable (x in the above example) and the relation is checked. The case else clause is executed if none of the above conditions match.

For statement

The for loop is a convenient way of performing a task a number of times, particularly when the number of iterations is known beforehand. For example:

```
for i = 1 to 10
  print "i is "; I
next i
```

In a for loop, the control variable (i in the above case) can be either increasing or decreasing. You may also specify a step value if you want the control variable to change by a value other than one. Here is a loop that prints the odd numbers from 9 to 1 backwards:

```
for i = 9 to 1 step -2
  print i
next
```

In the next statement that ends a for loop, it is not necessary to name the control variable as in the above example. If the control variable is specified it must match the most recent for loop. For example, the following is illegal:

```
for i = 1 to 5
  for j = 1 to 10
    print i, j
  next i         ' error, i not last loop
next j           ' j invalid here as well
```

When statement

The when statement is a structured way of executing statements at the time a particular event occurs. There are a number of different events which can trigger a when statement:

- a time duration has elapsed
- a certain time of day occurs
- a sequence of characters is received from the communications port

The structure of a when statement is very much like the structure of an if statement. For example,

```
when match "press enter" do
  send
end when
```

This will cause the send statement to be executed whenever the characters “press enter” are received from the communications port. A when statement remains active and can be triggered again until it is explicitly cleared with a clear when statement.

Subroutines and Functions

Subroutines and functions can be declared to help perform repetitive tasks. Here is a simple subroutine declaration:

```
sub test
  print "in sub test"
end sub
```

This subroutine can be called in one of two ways:

```
test
call test
```

The call keyword is optional here. Arguments can also be passed to functions, for example:

```
sub test(a as integer, b as integer)
  print "the sum of "; a; " and "; b; " is "; a+b
end sub

call test(1, 2)
call test(6, 11)
test (3, 4)
```

The output of this program is:

```
the sum of 1 and 2 is 3
the sum of 6 and 11 is 17
the sum of 3 and 4 is 7
```

Parameters are normally passed by reference, which means that if a subroutine changes the value of a variable, the change will be reflected in the variable which was used in the original parameter list. There are two exceptions to this rule. The first is when an expression is passed to a procedure. For example,

```
sub bump(a as integer)
  a = a + 1
end sub

dim x as integer
x = 3
print x
bump x
print x
bump x+1
print x
```

The output of this program is 3, 4, 4, 4. In the first call to bump, x is actually incremented because it is directly passed to the variable a in the bump subroutine. In the second call to bump, x is not incremented because it is not directly passed to the subroutine — x+1 is passed instead.

The second exception to the pass-by reference rule occurs when the BYVAL keyword is used. Refer to the BYVAL keyword for information and examples.

Functions are similar to procedures, except that they always return a value and this return value can be used in expressions. For example,

```
function f(a as integer, b as integer)
  f = a * b
end function
```

This declares a function which multiplies its arguments and returns their product. It can be used like this:

```
print f(5, 6)
x = y + f(z, w)
```

The first line prints the value 30, and the second line will first multiply z and w, then add that to y, and assign the result to x.

Script Command Reference

The following is a complete description of all available *QmodemPro for Windows 95* script commands. Script commands are presented in the following format:

Script Command

A brief description of the command.

```
Syntax

A description of the command's syntax.
```

Remarks

A more detailed description of the command, including parameters and their meaning, and how the command interacts with the rest of the script and the system in general.

See also

Related script commands.

Example

```
Most examples will be a complete script program that you can type in and run.
```

Click on the command you are interested in.

+ (Concatenation Operator

Concatenates (combines) two or more strings into one string.

Syntax

```
string1 + string2 [+ string3 ...]
```

Remarks

This operator is used to combine strings into a single, longer string. For instance, you can concatenate DATE and TIME variables into a single string variable called NOW (see example).

No spaces or other characters are placed between the strings during concatenation.

See also

LEFT, MID, RIGHT

Example

This example combines the current date and time strings into one string called now, and prints it on the screen.

```
dim now as string

now = date + " " + time

print now
```

$INCLUDE Directive

Includes another source file into the current source file.

Syntax

```
'$INCLUDE 'filename'
```

Remarks

The $INCLUDE directive is an instruction to the script compiler to include the named script file into the current file. This is useful if you have a standard set of subroutines and functions that you want to share between multiple scripts.

Example

```
[in file a.scr]
'$include 'b.scr'
call test

[in file b.scr]
sub test
  print "Hello, world!"
end sub
```

ABS Function

Returns the absolute value of a numeric expression.

Syntax

```
ABS(number)
```

Remarks

number can be any valid numeric expression.

The absolute value of a number is the quantity of the number without regard for its sign (positive or negative). A negative number has the same absolute value as the corresponding positive number. In other words,

abs(x) = x	if x is positive
abs(x) = -x	if x is negative

See also

SGN

Example

This example shows how two different numbers can have the same absolute value.

```
dim i as integer, j as integer

i = 5

j = -5

if abs(i) = abs(j) then

  print "i and j have the same absolute value"

end if
```

ACTIVATE Statement

Used to make *QmodemPro for Windows 95* the active application on the Windows 95 desktop.

Syntax

```
ACTIVATE
```

Remarks

This command will also restore the *QmodemPro for Windows 95* window if it is currently minimized.

See also

MINIMIZE, MAXIMIZE, MOVE, SIZE

Example

This example minimizes the *QmodemPro for Windows 95* window, waits for the string “connect” from the modem, then reactivates the application.

```
minimize

waitfor "connect"

activate
```

ADDLFTOCR Function

Used to get or set the current state of the "AddLFtoCR" toggle.

Syntax

```
ADDLFTOCR
```

or

```
ADDLFTOCR(onoff)
```

Remarks

There are two forms to the ADDLFTOCR function. The first form takes no arguments and simply returns the current setting. The second form sets the tate of the toggle and returns the previous state.

In both the parameter and the return value, a nonzero value means line feeds are added to incoming carriage returns. A zero value means no additional characters are added.

See also

DUPLEX

Example

This example shows how the ADDLFTOCR function can be used in a logon script to turn on the line feed control.

```
dim oldcrlf as integer

oldcrlf = addlftocr(on)

waitfor "UserID"

send "123456"

addlftocr oldcrlf
```

ADDPHONEENTRY Statement

The ADDPHONEENTRY statement is used to add a new entry to the phonebook.

Syntax

```
ADDPHONEENTRY phoneentry
```

Remarks

The name of every entry in the phonebook must be unique. If the entry name already exists, this statement will fail. The *phoneentry* variable is a variable of type PHONEENTRY. Refer to the appendix for the phoneentry declaration.

See also

DIAL

Example

This example shows howyou might set up the MSI HQ BBS as a phonebook entry and then reset the downloads and uploads to zero.

```
dim entry as phoneentry
entry.NAME       = "MSI HQ BBS"
entry.AREACODE   = "805"
entry.TAPIDEVICE = "Courier HST Dual Standard Fax+ASL"
entry.NIMBER (1) = "873-2400"
entry.USERID     = "Canfield Customer"
entry.PASSWORD   = "QMPROWIN95"
entry.UPLOADS    = 0
entry.DOWNLOADS  = 0
entry.CONNECTTYPE = 0 //Set data as connect type
addphoneentry (entry)
```

AND Operator

AND performs a bitwise AND operation between its operands.

Syntax

```
op1 AND op2
```

Remarks

op1 and *op2* are logical expressions or numeric values of any integral type (byte, integer, or long integer).

This is the logical truth table for the AND operator:

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

See also

EQV, IF Statement , IMP, NOT, OR Operator , XOR

Example

The first part of this example shows how the AND operator can be used as a bitwise logical operator to perform manipulations on integers. The second part of this example shows how the AND operator can be used to combine the results of two comparisons and supply the result to an IF statement.

```
dim i as integer
i = 129
if i and 127 = 1 then
  print "yes"
end if

dim j as integer
j = 4
if i = 129 and j = 4 then
  print "yes again"
end if
```

ASC Function

Returns the ASCII value (from 0 to 255) of the first character in a string.

Syntax

```
ASC(string)
```

Remarks

string is any string expression. The return value of this function is the ASCII value of the first character in the string. If the string is empty this function returns zero.

The opposite of the ASC function is the CHR function. The CHR function returns the character corresponding to a particular ASCII code.

See Also

CHR, VAL

Example

This example prints 104, which is the ASCII value of “h”, the first character in the string “hello”.

```
dim a as string

a = "hello"

print asc(a)
```

ATN Function

Returns the arctangent of a numeric expression.

Syntax

```
ATN(number)
```

Remarks

The arctangent of *number* is the angle in radians whose tangent is equal to *number*. You can convert radians to degrees by multiplying by 180/pi (pi is approximately 3.14159).

See Also

COS, SIN, TAN

Example

This example uses the atn function to compute the value of pi.

```
print 4*atn(1)
```

AUTOANSWER Statement

Used to turn the AutoAnswer mode on or off for a specific modem.

Syntax

```
AUTOANSWER onoff
```

Remarks

If a modem name is passed to this function, autoanwser modem is turned **ON** for that modem. The return value is TRUE if the operation is successful. If AUTOANSWER OFF is used, auto answer mode is turned **OFF**.

See Also

DIAL, HANGUP

Example

This example tells you whether a key was pressed or a call was answered.

```
if autoanswer (getmodemname (0) ) then

  print "Autoanswer is now on."

else

  print "Failed to configure modem for AUTOANSWER."
```

BEEP Statement

Causes *QmodemPro* to sound the regular Windows 95 beep sound.

Syntax

```
BEEP
```

Remarks

If you have attached a .wav file to the “Default Beep” sound in Windows 95 Control Panel, then this command will play the corresponding .wav file.

See Also

SOUND

Example

This example simply sounds the Windows 95 default beep.

```
beep
```

BREAK Statement

Sends a break signal to the communications port.

Syntax

```
BREAK
```

Remarks

A break is a special signal sent to the communications port. The break signal is often used with older communications hardware, such as mainframes. It usually does not have any effect when connected to a bulletin board or online service.

See Also

HANGUP

Example

This example sends a break signal, waits for the string “connection lost”, then ends the script.

```
break

waitfor "connection lost"

end
```

BYTE Type

Used to declare a variable that can handle byte sized numbers.

Remarks

Variables of byte type can hold values that range from 0 to 255.

See Also

DIM, INTEGER, LONG, REAL, TYPE

Example

This example declares a variable of type BYTE and assigns a value to it.

```
dim i as byte

i = 5

print i
```

BYVAL Keyword

Indicate the parameter passing method for Subs and Functions.

Syntax

```
Function Name(BYVAL tStr As String) As String

Sub Name(BYVAL tNum As Long)
```

Remarks

The **ByVal** keyword allows you to define how a parameter to be handled within a subroutine at the time of designing the **Sub** or **Function**. The default method for handling parameters is to pass by reference, which means that the subroutine or function can modify the value of the original variable.

By using **ByVal**, it will force the compiler to create a working copy of the data type and leave the passed parameter untouched on return.

See Also

FUNCTION, SUB

Example

```
Rem The difference in passing by value or by reference


Dim tStr As String
Function Extend(aStr As String, aLen As Word) As String
  If Len(aStr) < aLen Then
    aStr = aStr + String(aLen - Len(AsTr), ".")
  End If
  Extend = aStr
End Function


Function ExtendBV(ByVal aStr As String, aLen As Word) As String
  If Len(aStr) < aLen Then
    aStr = Pad(aStr, aLen)
  End If
  ExtendBV = aStr
End Function


tStr = "Testing this here String!"
Print "["; ExtendBV(tStr, 60); "]"
Print "["; tStr; "]" : Rem ByVal = no change
Print "["; Extend(tStr, 60); "]"
Print "["; tStr; "]" : Rem By reference = change
```

CALL Statement

Executes a subroutine or function.

Syntax

```
CALL name[( arg [, arg ...])]
```

or

```
name [arg[, arg ...]]
```

Remarks

name is the name of the subroutine or function to execute. Subroutines and functions must be at least declared before you can call them (see the DECLARE statement for information on declaring subroutines and functions).

arg is an argument that is passed to the sub-program. Multiple arguments are separated with commas. When using the first syntax with the CALL keyword, parentheses are required around the argument list. When using the second syntax, omitting the CALL keyword, parentheses are optional around the parameter list.

Arguments are normally passed to the subroutine or function in “reference” mode. This means that if the corresponding argument in the function is changed, the original copy will be changed too. This behavior may be changed by passing an expression to the function, like a+1 or (a) (the parentheses around a serve only to create an expression and prevent a subroutine from changing its value). See the example below for an illustration of how this works.

See Also

DECLARE, FUNCTION, GOSUB, SUB

Example

This example declares a subroutine called f which takes two arguments. The subroutine adds one to the first argument, and adds two to the second argument. If you run this script, you will notice that the value of a in the main program is changed, while the value of b remains the same. This is because b is not directly passed to the subroutine, but a copy of b is. The subroutine changes the copy and does not affect the actual value of b.

```
declare sub f(x as integer, y as integer)

dim a as integer, b as integer

a = 4

b = 7

call f(a, (b))

print "a is "; a

print "b is "; b


sub f(x as integer, y as integer)

  x = x + 1

  y = y + 2

end sub
```

CAPTURE Statement

Used to open or close a terminal capture file.

Syntax

```
CAPTURE filename

CAPTURE ON

CAPTURE OFF
```

Remarks

filename is the name of the file to which captured data will be appended. Since the file name is a string, it will normally be enclosed in quotation marks.

on turns on the capture file specified in **Options/Files/File Definitions**.

off closes any currently open capture file.

If you specify a new capture file while another capture file is still open, the first capture file will be closed before the new one is opened.

If the capture file cannot be opened, the ERR_FILEOPEN error will be generated. This error can be caught with the CATCH statement.

See Also

CATCH, PRINTER

Example

This example opens the capture file “test.cap”, sends an Enter to the communications port, waits 10 seconds, then closes the capture file.

```
capture "test.cap"

send

delay 10

capture off
```

CARRIER Function

Determines whether the modem is currently online and connected to another modem.

Syntax

```
CARRIER
```

Remarks

The CARRIER function returns TRUE if the modem currently reports that it is online and connected to another modem. If not, it returns FALSE. The return value of this function corresponds with the state of the “Online/Offline” indicator at the bottom of the terminal window.

See Also

HANGUP

Example

This example sends the string “bye” to the communications port, then waits until the modem reports that carrier is no longer active (that is, the remote modem has hung up).

```
send "bye"

while carrier do

wend
```

CASE Statement

Introduces a new case in a SELECT CASE statement. Please see the description of the SELECT CASE statement for more information and examples.

CATCH Statement

Used to catch runtime errors and perform error recovery actions.

Syntax

```
CATCH errvalue [, errvalue ...]
```

```
or
```

```
CATCH ALL
```

Remarks

The CATCH mechanism provides a convenient way of responding to errors that may occur while your script is running. The errvalue values must be one of the following:

ERR_ARRAYSUBSCRIPT

Caused by accessing an array using an invalid subscript index.

ERR_FILEOPEN

An error during a file open operation causes this error. The commands that can cause this exception are CAPTURE, OPEN, and SHELL.

ERR_FILERENAME

Caused by an error during a file rename operation. This could be caused by the original file not being found, or open by another application, or the destination filename is already in use. The NAME statement can cause this error.

ERR_FUNCTIONNOTFOUND

Caused by trying to call a function in a dynamic link library (DLL) where the named function does not exist.

ERR_INVALIDFILENUMBER

Caused by using an invalid file number in any file operation, including OPEN, CLOSE, PRINT, INPUT, INKEY, LOF, and so on.

ERR_LIBRARYNOTFOUND

Caused by trying to call a function in a dynamic link library (DLL) where the named DLL does not exist.

ERR_MATH

Caused by trying to divide by zero, taking the logarithm of zero or a negative number, or by taking the square root of a negative number.

ERR_PATH

Caused by an invalid drive or path name in one of the following commands: CHDIR, CHDRIVE, MKDIR, RMDIR.

ERR_TIMEOUT

Caused by a timeout during one of the following commands: INPUT, RECEIVE, WAITFOR.

A CATCH statement may only be placed at the end of a user defined subroutine or function, or at the end of the main program body. During normal operation (the case where no runtime error occurs) the statements after the CATCH statement are skipped. If one of the above runtime errors occurs, the CATCH block for the currently executing function will be searched for a handler for the error. If there is no CATCH block or if there is no specific CATCH handler for the error that occurred, control will return to the function that called the current function. Its CATCH block (if any) will be searched for a handler, and so on up the call chain. If the error propagates all the way up to the main program body and there is no CATCH handler there for the error, then the script is automatically halted with an appropriate error message.

See Also

ERROR

Example

This example defines a subroutine that opens a file called “test.dat”, reads the first line of text from the file, and closes it. If there is an error opening the file, the subroutine prints an error message.

```
declare sub test
dim a as string
call test(a)
print a

sub test(s as string)
  open "test.dat" for input as #1
  input #1, s
  close #1
catch err_fileopen
  print "could not open test.dat"
end sub
```

CHAIN Statement

Used to transfer control to another script.

Syntax

```
CHAIN scriptname
```

Remarks

This command terminates execution of the current script and starts execution of the named script. All currently open files are closed and variables are discarded.

The script named in this command must be the name of a compiled script with the proper .QSX extension. Scripts are not automatically compiled by this command.

See Also

END, STOP

Example

This example shows two files, a.QSC and b.QSC, and demonstrates how you can start execution of the second script from the first.

```
[in file a.QSC]

print "in a.QSC"

chain "b.QSX"


[in file b.qsc]

print "in b.qsc"
```

CHDIR Statement

Changes the current directory.

Syntax

```
CHDIR directory
```

Remarks

This command changes to the directory named in the *directory* argument.

As with the DOS CHDIR command, this does not change the current drive, even if it is specified in the argument. However, if you change the current directory for a drive that is not the current drive, the change will be remembered until you next change to the new drive using the CHDRIVE command.

See Also

CHDRIVE, CURDIR, CURDRIVE, MKDIR, RMDIR

Example

This example shows various ways the CHDIR command can be used.

```
chdir "\"               'change to the root directory
chdir "c:\qmwin"        'change to C:\qmwin
dim newdir as string
newdir = "c:\temp"      'assign specified directory to
                        'string variable NEWDIR
chdir newdir            'change to the new directory
```

CHDRIVE Statement

Changes the current drive.

Syntax

```
CHDRIVE driveletter
```

Remarks

This command changes the current drive to *driveletter*. If *driveletter* is longer than one character, only the first character in the string is used.

See Also

CHDIR, CURDIR, CURDRIVE

Example

This example takes advantage of the fact that only the first letter of the argument to CHDRIVE is used, and changes to the directory “c:\test” no matter which drive this script was started from.

```
dim a as string

a = "c:\test"

chdrive a

chdir a
```

CHR Function

Returns the ASCII character corresponding to the specified ASCII code value in the range of 0 to 255.

Syntax

```
CHR(number)
```

Remarks

This function returns the ASCII character corresponding to *number.*

The opposite of the CHR function is the ASC function. The ASC function returns the ASCII character value of the first character in a string.

See Also

ASC, SPACE, STRING

Example

This example prints “Hi” using the ASCII values of the characters “H” and “i”.

```
print chr(72); chr(105)
```

CLOSE Statement

Closes a file or files opened with the OPEN statement.

Syntax

```
CLOSE [[#]filenumber[, [#]filenumber]...]
```

Remarks

filenumber is the number of an open file. The file associated with the given file number will be closed.

CLOSE without any parameters will close all open files.

Although *QmodemPro for Windows 95* will automatically close files when the script terminates, you should always close any files that you open. There are only a limited number of file numbers available; if you run out of file numbers you will not be able to open any more files.

See Also

OPEN, RESET

Example

This file opens a new file called “test.txt” for output, writes a line of test data to the file, then closes the file.

```
open "test.txt" for output as #1

print #1, "test data"

close #1
```

CLOSEPORT Statement

This statement closes the currently opened port, if any.

Syntax

```
CLOSEPORT
```

Remarks

CLOSE without any parameters will close all open ports.

See Also

OPENSERIALPORT, OPENTCPIPPORT

Example

Here is an example of how you might open COM1, send a dial command, and then close the port.

```
if openserialport ("COM1") then print "COM 1 opened."

SEND "ATDT 1-805-873-2400 ^M"

Closeport

print "port closed."
```

CLS Statement

Clears the terminal screen and returns the cursor to the top left corner. This command also resets the current text color to the default text color.

Syntax

```
CLS
```

Remarks

This command is similar to the DOS CLS “clear screen” command.

See Also

COLOR

Example

This example clears the screen and writes “Hello, world!” in the upper left hand corner.

```
cls

print "Hello, world!"
```

COLOR Statement

Sets the current terminal color to the specified foreground and background.

Syntax

```
COLOR foreground [, background]
```

Remarks

Sets the current terminal color to the specified foreground and background values. If the background value is not specified, then it remains unchanged.

The color values used by this command are listed in the appendix of this manual.

See Also

CLS, SCREEN

Example

This example sets the color to yellow on blue and prints a text string in that color.

```
color 11, 4

print "yellow text on blue background"
```

CONFIGCAPTUREFILE Function

Syntax

```
CONFIGCAPTUREFILE
```

Remarks

This function returns the exact text contained in the Options/Files/File definitions dialog entry for the capture file name. This is usually the fully qualified path and filename of the capture file, but may return only the filename if no path was entered in the config dialog.

See Also

CONFIGDOWNLOADPATH, CONFIGLOGFILE, CONFIGSCRIPTPATH, CONFIGSCROLLBACKFILE, CONFIGTRAPFILE, CONFIGUPLOADPATH

Example

```
UPLOAD (CONFIGCAPTUREFILE, Zmodem)
```

CONFIGDOWNLOADPATH Function

Syntax

```
CONFIGDOWNLOADPATH
```

Remarks

This function returns the exact text contained in the Options/Files/Path definitions dialog entry for the download path. This is the fully qualified path for the download directory, without a trailing backslash.

See Also

CONFIGCAPTUREFILE, CONFIGLOGFILE, CONFIGSCRIPTPATH, CONFIGSCROLLBACKFILE, CONFIGTRAPFILE, CONFIGUPLOADPATH

Example

```
CHDIR CONFIGDOWNLOADPATH
```

CONFIGLOGFILE Function

Syntax

```
CONFIGLOGFILE
```

Remarks

The function returns the exact text contained in the Options/Files/File definitions dialog entry for the logfile name. This is usually the fully qualified path and filename of the log file, but may return only the filename if no path was entered in the config dialog.

See Also

CONFIGDOWNLOADPATH, CONFIGCAPTUREFILE, CONFIGSCRIPTPATH, CONFIGSCROLLBACKFILE, CONFIGTRAPFILE, CONFIGUPLOADPATH

Example

```
UPLOAD (CONFIGLOGFILE, Zmodem)
```

CONFIGSCRIPTPATH Function

Syntax

```
CONFIGSCRIPTPATH
```

Remarks

The function returns the exact text contained in the Options/Files/Path definitions dialog entry for the scripts path. This is the fully qualified path for the scripts directory, with no trailing backslash.

See Also

CONFIGDOWNLOADPATH, CONFIGCAPTUREFILE, CONFIGLOGFILE, CONFIGSCROLLBACKFILE, CONFIGTRAPFILE, CONFIGUPLOADPATH

Examples

```
CHDIR CONFIGSCRIPTPATH
```

CONFIGSCROLLBACKFILE Function

Syntax

```
CONFIGSCROLLBACKFILE
```

Remarks

The function returns the exact text contained in the Options/Files/File definitions dialog entry for the scrollback filename. This is usually the fully qualified path and filename of the scrollback file, but may return only the filename if no path was entered in the config dialog.

See Also

CONFIGDOWNLOADPATH, CONFIGLOGFILE, CONFIGSCRIPTPATH, CONFIGCAPTUREFILE, CONFIGTRAPFILE, CONFIGUPLOADPATH

Example

```
UPLOAD (CONFIGSCROLLBACKFILE, Zmodem)
```

CONFIGTRAPFILE Function

Syntax

```
CONFIGTRAPFILE
```

Remarks

The function returns the exact text contained in the Options/Files/File definitions dialog entry for the trap filename. This is usually the fully qualified path and filename of the trap file, but may return only the filename if no path was entered in the config dialog.

See Also

CONFIGDOWNLOADPATH, CONFIGLOGFILE, CONFIGSCRIPTPATH, CONFIGSCROLLBACKFILE, CONFIGCAPTUREFILE, CONFIGUPLOADPATH

Example

```
UPLOAD (CONFIGTRAPFILE, Zmodem)
```

CONFIGUPLOADPATH Function

Syntax

```
CONFIGUPLOADPATH
```

Remarks

The function returns the exact text contained in the Options/Files/File definitions dialog entry for the upload file area path. This is the fully qualified path for the upload path, with no trailing backslash.

See Also

CONFIGDOWNLOADPATH, CONFIGLOGFILE, CONFIGSCRIPTPATH, CONFIGSCROLLBACKFILE, CONFIGTRAPFILE, CONFIGCAPTUREFILE

Examples

```
CHDIR CONFIGUPLOADPATH
```

CONST Statement

Used to assign symbolic names that will be used in place of actual values.

Syntax

```
CONST name = expression [, name = expression ...]
```

Remarks

name is the name of the new constant.

expression is the value to assign to the symbolic constant name.

The type of name is determined by the type of the expression.

Constants defined in subroutines or functions can only be used within the subroutine or function. Constants defined with a CONST statement in the main program body can be used throughout the program.

Example

This example declares the constant “myname” and assigns a value to it.

```
const myname = "John Doe"

print "My name is "; myname
```

COPYFILE Function

Copy a file to another directory or drive.

Syntax

```
COPYFILE(SourceFile, TargetFile)
```

Remarks

The **CopyFile** function does just as the name describes, it copies a file from one location to another. It will issue no warning, prior to over-writing a file with an identical name, if one exists in the target directory. The function will return either **True** or **False** upon the success of the operation. Both *SourceFile* and *TargetFile* must include the file name as a path-only in *TargetName* will be rejected.

See Also

NAME

Example

```
Rem Copy a file across drives


If CopyFile("C:WCLIST.OUT", "D:\BAK\DATA\WCLIST.BAK") Then
    Print "File copied"
Else
    Print "Copy failed"
    Beep
    WaitEnter
End If
```

COS Function

Returns the cosine of an angle.

Syntax

```
COS(angle)
```

Remarks

angle is the measurement of an angle expressed in radians. You can convert radians to degrees by multiplying by 180/pi (pi is approximately 3.14159).

See Also

ATN, SIN, TAN

Example

This example prints the cosine of 1 radian.

```
print cos(1)
```

CSRLIN Function

Returns the current vertical coordinate position (row number) of the cursor.

Syntax

```
CSRLIN
```

Remarks

This value is usually an integer in the range 1 through 25, but may be larger depending on the number of lines set in the **Options/Emulations** dialog.

See Also

LOCATE, POS

Example

This example clears the screen, then prints the cursor line twice. The first time it will be 1 since the cursor is on the top line of the screen, and the second time it will be 2 (the cursor moved down because of the first print statement).

```
cls

print csrlin

print csrlin
```

CURDIR Function

Returns the current drive and directory.

Syntax

```
CURDIR
```

Remarks

The current directory of the current drive is returned without a trailing backslash (unless the current directory is the root directory).

See Also

CHDIR, CHDRIVE, CURDRIVE, MKDIR, RMDIR

Example

This example prints the current directory for the current drive.

```
print "The current directory is "; curdir
```

CURDRIVE Function

Returns the current drive letter.

Syntax

```
CURDRIVE
```

Remarks

The current drive letter is returned as an uppercase letter. This is the same drive that is returned by the CURDIR function.

See Also

CHDIR, CHDRIVE, CURDIR, MKDIR, RMDIR

Example

This example prints the current drive letter.

```
print "The current drive is "; curdrive
```

DATE Function

Returns the current date as a string.

Syntax

```
DATE
```

Remarks

The date is returned in the system format specified in Windows 95 Control Panel, Regional Settings section, short date format.

See Also

TIME

Example

This example prints today’s date.

```
print "Today is "; date
```

DATETIMEDIFF Function

Computes the difference between two variables of type DateTime and returns the result in Days and Seconds.

Syntax

```
DATETIMEDIFF (dt1 as DateTime, dt2 as DateTime, days as integer, seconds as integer)
```

Remarks

The difference is computed without regard to which datetime (dt1 or dt2) is greater. The result will always be a positive number.

See Also

DATETIME

Example

This exmple prints to the screen the number of days and seconds between dialing a BBS and hanging up.

```
dim starttime as datetime
dim endtime as datetime
dim totaldays as integer
dim totalseconds as integer

getcurrentdatetime (starttime)
send "ATDT 1-805-873-2400^M"
waitfor "NO CARRIER"
getcurrentdatetime (endtime)
datetimediff (starttime,endtime,totaldays,totalseconds)
print "You were logged on for ";totaldays;"days \
  and ";totalseconds;" seconds."
```

DECLARE Statement

Allows you to declare subroutines and functions before their actual definition. Also allows declaration of DLL subroutines and functions.

Syntax

```
DECLARE [FUNCTION | SUB] name [LIB "libname" [ALIAS "aliasname"]] [(argument list)]
[AS returntype]
```

Remarks

name is the name of the subroutine or function.

The argument list lists the parameters to the subroutine or function. This argument list must match the argument list declared in the actual definition of the subroutine or function.

The LIB and ALIAS clauses are used to declare a function that actually exists in another DLL. For more information on declaring and using DLL functions, see Using DLL Functions.

The final AS clause is used to declare the return type of a function.

Functions must be declared before they can be used. If the function definition appears later in the script source file from where you want to call the function, the DECLARE statement can be used to declare the function before the call.

See Also

CALL, FUNCTION, SUB

Example

This example uses the DECLARE statement to declare a function so it can be used before its actual definition appears later in the file.

```
declare function timestwo(x as integer)
print timestwo(5)

function timestwo(x as integer)
  timestwo = x * 2
end function
```

DEL Statement

Deletes a file from disk.

Syntax

```
DEL filename
```

Remarks

filename is the name of the file to delete. Wildcards * and ? are not supported.

This statement is identical to the script KILL command.

You cannot delete an open file, whether it is has been opened by your script, *QmodemPro for Windows 95*, or another application.

See Also

KILL, RMDIR

Example

This example deletes the file “test.dat” from the current directory.

```
del "test.dat"
```

DELAY Statement

Used to suspend script execution for a certain time interval.

Syntax

```
DELAY time
```

Remarks

time is the amount of time to suspend execution, expressed in seconds. If you want to delay for less than a second, use ordinary decimal notation.

This command is identical to the script PAUSE command.

See Also

PAUSE, WAITFOR, WHEN QUIET, WHEN TIME

Example

This example uses the “atz” command to reset a Hayes-compatible modem, waits for half a second, then sends a command to dial a telephone number.

```
send "atz"

delay 0.5

send "atdt5551212"
```

DIAL Statement

This statement is used to dial an entry or entries from the phonebook.

Syntax

```
DIAL ENTRY number
or
DIAL GROUP groupname
or
DIAL SEARCH string
or
DIAL MANUAL number
```

Remarks

There are four forms to the DIAL command:

DIAL ENTRY

This form allows you to dial a specific entry number in the phonebook.

DIAL GROUP

This form allows you to dial all the entries in a named group.

DIAL SEARCH

This form allows you to search for a string and dial all entries that contain that string.

DIAL MANUAL

This form allows you to dial a specific phone number from the script.

After dialing a number from a script in this way, the script file specified in the dialing directory (if any) is not executed. After connect the script proceeds from the next statement after the DIAL command.

See Also

ADDPHONEENTRY, DIALNEXT, HANGUP, LASTCONNECTUSERID, LASTCONNECTPASSWORD,

Examples

This example shows each of the four ways the DIAL command can be used.

```
dial entry 3
dial group "Morning mail"
dial search "Mustang"
dial manual "555-1212"
```

DIALNEXT Function

This function is used to dial the next entry in a group after using the DIAL GROUP command.

Syntax

```
DIALNEXT
```

Remarks

When using the DIAL GROUP command, all the entries in the phonebook corresponding to the dialed group are marked for dial.   After connecting to an entry, use the DIALNEXT function to continue dialing the remaining marked entries in the phonebook.

This function returns FALSE if there are no further marked entries in the phonebook, otherwise it returns TRUE.

See Also

ADDPHONEENTRY, DIAL, HANGUP

Examples

This example shows how the DIALNEXT command is used in conjunction with DIAL GROUP.

```
dial group "Morning mail"

do

  ... do whatever needs to be done online ...

loop while dialnext
```

DIALOG Statement

This statement is used to declare a group box style.

Syntax

```
DIALOG dialogtype x, y, w, h
  [CAPTION caption]
  [FONT size, fontname]
  [integer-field AS CHECKBOX title, id, x, y, w, h]
  [integer-field AS COMBOBOX id, x, y, w, h]
  [CTEXT title, id, x, y, w, h]
  [DEFPUSHBUTTON title, id, x, y, w, h]
  [string-field AS EDITTEXT id, x, y, w, h]
  [GROUPBOX title, id, x, y, w, h]
  [integer-field as LISTBOX id, x, y, w, h]
  [LTEXT title, id, x, y, w, h]
  [PUSHBUTTON title, id, x, y, w, h]
  [integer-field AS RADIOBUTTON title, id, x, y, w, h]
  [RTEXT title, id, x, y, w, h]
  ...
END DIALOG
```

Remarks

The DIALOG statement declares a *dialog template*. A dialog template is much like a user defined type in that it contains named fields in which you can place information. Dialog templates can also have a number of unnamed fields which serve to place extra information in the Windows 95 dialog that is created based on this template.

Dialog boxes are discussed in depth in the section titled Using Dialog Boxes.

DIALOGBOX Function

Displays and executes a dialog box based on a dialog box template.

Syntax

```
DIALOGBOX(dialogvar)
```

Remarks

This function creates and executes a dialog box based on the dialog box variable. Its return value depends on the event that caused the dialog box to close. In most cases this will be either IDOK or IDCANCEL depending on whether the user pressed the OK button or the Cancel button to close the dialog box. In more advanced cases the meaning of the return value will be user defined.

The dialog box variable used in this function should not be already active.

Dialog boxes are discussed in depth at the end of this chapter, in the section titled Using Dialog Boxes.

See Also

DIALOG

DIM Statement

The DIM statement is used to declare variables of any type including array types. An array is a variable containing a series of values that are all of the same type.

Syntax

```
DIM var[([lowerbound TO] upperbound)] AS type[, var[([lowerbound TO] upperbound)] AS
type...]
```

Remarks

var is the name of the array or variable being declared.

lowerbound and upperbound declare the lowest and highest subscript values that are allowed if an array is being declared. If lowerbound is omitted, it defaults to zero.

Each variable must have an associated type declaration with the AS clause.

Variables declared within a subroutine or function are only available from within that subroutine or function. Variables declared in the main program body are available throughout the entire script file.

An array consists of a number individual variables, called “elements”, which are referred to by numbers, called “subscripts” indicating the position of each element in the array.

See Also

STATIC

Example

This example declares a simple integer “i” and an array “a” which refers to a sequence of six integers, numbered a(0) through a(5).

```
dim i as integer, a(5) as integer
for i = 1 to 5
  a(i) = i*2
next
for i = 1 to 5
  print a(i)
next
```

DO ... LOOP Statement

Repeatedly executes a block of statements while (or until) a specified condition is met.

Syntax

```
DO [{WHILE | UNTIL} expression]
  [statements]
LOOP
```

or

```
DO
  [statements]
LOOP [{WHILE | UNTIL} expression]
```

Remarks

The first example above tests for the specified condition at the beginning of the loop, and exits the loop when the condition is met.

The second example tests for the specified condition at the end of the loop, and continues until the condition is met.

expression is any logical expression that evaluates to either true (nonzero) or false (zero). The keyword WHILE repeats the loop while the expression remains true. The UNTIL keyword repeats the loop until the expression becomes true.

statements are program statements that are repeated. Note that statements are optional; a loop can be empty, waiting for an external event such as a keypress or an incoming character.

Every DO statement in a program must have a corresponding LOOP, and each LOOP must have a preceding DO.

See Also

EXIT DO, FOR ... NEXT, WHILE ... WEND

Example

This example prints out the integers from 1 through 5 using a DO WHILE ... LOOP statement.

```
dim i as integer
i = 0
do while i < 5
  i = i + 1
  print i
loop
```

DOORWAY Function

Used to get or set the current state of the Doorway toggle.

Syntax

```
DOORWAY
```

or

```
DOORWAY(onoff)
```

Remarks

There are two forms to the DOORWAY function. The first form takes no arguments and simply returns the current setting. The second form sets the state of the toggle and returns the previous state.

In both the parameter and the return value, a nonzero value means Doorway mode is turned on.   A zero value means Doorway mode is turned off.

When turned on, Doorway mode causes all keystrokes that can be typed on the keyboard to be sent directly to the remote.   This will work only with a remote host that understands the Doorway keystrokes.

Example

This example shows how to save the doorway setting, change it, and then restore it.

```
dim olddoorway as integer

olddoorway = doorway(on)

  ...

doorway olddoorway
```

DOWNLOAD Function

Used to receive files from a remote computer.

Syntax

```
DOWNLOAD(filename, protocol)
```

Remarks

This function initiates a file transfer to receive files from a remote computer. At the time this command is executed, the remote computer must already have started the file transfer. If you are connecting to a bulletin board system (BBS) then your script should already have sent the command to the BBS that will start the file transfer.

The protocol is one of the following predefined constants:

ASCII, XMODEM, XMODEMCRC, XMODEM1K, XMODEM1KG, YMODEM, YMODEMG, ZMODEM, KERMIT

For the first five protocols, the filename parameter must specify an actual file name in which to place the received file. The ASCII and Xmodem variant protocols do not supply a filename, so one must be supplied in the DOWNLOAD function.

For the last four protocols, the filename parameter should be the name of a directory in which to place the received files. If this parameter is an empty string (" ") then the download directory specified in **Options/Files/Path Definitions** will be used.

The DOWNLOAD function returns zero if the file transfer was successful. If the transfer was unsuccessful, DOWNLOAD returns the error code describing the error. For a list of error codes see Chapter 4.

This function is identical to the RECEIVEFILE function.

See Also

RECEIVEFILE, SENDFILE, UPLOAD

Example

This example receives a file from a remote computer, assuming that the remote computer has already started to send the file.

```
if download("", Zmodem) = 0 then
  print "file transfer ok!"
end if
```

DUPLEX Function

Used to get or set the current duplex (local echo) setting.

Syntax

```
DUPLEX
```

or

```
DUPLEX(onoff)
```

Remarks

There are two forms to the DUPLEX function. The first form takes no arguments and simply returns the current duplex setting. The second form sets the duplex and returns the previous state of the duplex setting.

In both the parameter and the return value, a nonzero value means full duplex or no local echo. A zero value means half duplex or local echo.

See Also

HOSTECHO

Example

This example shows how the DUPLEX function can be used to control the local echo of characters sent to the communications port.

```
dim oldduplex as integer

oldduplex = duplex(0)

send "this will be echoed locally"

duplex oldduplex

send "this will not be echoed locally"
```

EDITFILE Statement

Used to invoke the internal editor with a specified file.

Syntax

```
EDITFILE filename
```

Remarks

This function brings up the internal editor with the named file loaded ready for editing.   Note that that script continues to run after the editor is started.

See Also

VIEWFILE

Example

This example shows how the EDITFILE command might be used to edit the host mode user file.

```
editfile "host.usr"
```

ELSE Statement

The ELSE statement is used to introduce the alternative portion of an IF statement. See the discussion of the IF statement for more information.

ELSEIF Statement

The ELSEIF statement is used to introduce an optional alternative portion of an IF statement. See the discussion of the IF statement for more information.

EMULATION Statement

This statement is used to change the current terminal emulation.

Syntax

```
EMULATION emulation
```

Remarks

The emulation argument must be one of the following predefined constants:

ADDSVP60, ADM3A, ANSI, AVATAR, DEBUGASCII, DEBUGHEX, DG100, DG200, DG210, HAZELTINE1500, HEATH19, IBM3101, IBM3270, RIPSCRIP, TTY, TVI910, TVI912, TVI920, TVI922, TVI925, TVI950, TVI955, VIDTEX, VT100, VT102, VT220, VT320, VT52, WYSE30, WYSE50, WYSE60, WYSE75, WYSE85, WYSE100, WYSE185

See Also

DIAL

Example

This example changes the current terminal emulation to RIPscrip.

```
emulation ripscrip
```

END Statement

Ends a script and returns to normal terminal operation.

Syntax

```
END
```

Remarks

The END statement immediately terminates the currently executing script and returns to normal terminal operation. It may be executed at any time. Any currently open files are closed before the script ends.

See Also

STOP

Example

This example print a string to the terminal, then ends the script. The second print statement is never executed.

```
print "hello world"

end

print "this is never executed"
```

ENVIRON Function

Returns the value of the specified DOS environment variable.

Syntax

```
ENVIRON(envname)
```

Remarks

envname is the name of the environment variable to retrieve. The *envname* parameter should be in upper case. If the specified environment variable is not found, an empty string will be returned.

Example

This example prints out the current DOS PATH setting.

```
print environ("PATH")
```

EOF Function

Used to determine whether the end of a file has been reached.

Syntax

```
EOF(filenum)
```

Remarks

This function returns TRUE if the end of the file specified by the *filenum* parameter has been reached.

See Also

LOC, LOF, SEEK

Example

This example prints the contents of a text file line by line until the end of the file is encountered.

```
dim a as string
open "test.dat" for input as #1
do while not eof(1)
  input #1, a
  print a
loop
close #1
```

EQV Operator

EQV performs a bitwise equivalence operation between its operands.

Syntax

```
op1 EQV op2
```

Remarks

op1 and op2 are logical expressions or numeric values of any integral type (byte, integer, or long integer).

This is the logical truth table for the EQV operator:

a	b	a EQV b
0	0	1
0	1	0
1	0	0
1	1	1

See Also

AND, IMP, NOT, OR Operator , XOR

Example

This example shows how the EQV operator can be used in an IF statement.

```
dim i as integer, j as integer, k as integer
i = 10
j = 8
k = 6
if i > j eqv j > k then
  print "both expressions are true or both are false"
end if
```

ERROR Statement

Causes a predefined or user defined runtime error to occur.

Syntax

```
ERROR errvalue
```

Remarks

If there is a CATCH handler for the error value, then it will be executed. Otherwise, the script will be terminated automatically. See the CATCH statement for a list of predefined runtime error values.

See Also

CATCH

Example

This example forces the ERR_FILEOPEN error to occur, which will cause the script to jump immediately to the handler for ERR_FILEOPEN. As a result, the first print statement won't be executed.

```
error err_fileopen

print "this won't be printed"


catch err_fileopen

print "caught the error"
```

EXISTS Function

Determines whether a specific file exists on disk.

Syntax

```
EXISTS(filename)
```

Remarks

This function returns TRUE if the specified file exists on disk. DOS wildcards are not accepted.

See Also

FINDFIRST, FINDNEXT

Example

This example determines whether the “c:\autoexec.bat” file still exists.

```
if exists("c:\autoexec.bat") then
  print "autoexec.bat is still there"
end if
```

EXIT Statement

Aborts a loop, subroutine, or procedure without waiting for normal termination or return.

Syntax

```
EXIT [DO | FOR | FUNCTION | SUB]
```

Remarks

One of the four keywords above must be included in the EXIT statement.
Use the EXIT DO and EXIT FOR statements to exit a loop defined by a DO ... LOOP or a FOR ... NEXT statement.
If a loop is nested, the EXIT statement will stop only the current (innermost) loop.
Use the EXIT FUNCTION and EXIT SUB statements to abort procedures defined by a FUNCTION or SUB statement.
If you are exiting a function with EXIT FUNCTION and have not already assigned a value to the function result, the function will return zero or an empty string, depending on the type of the function.

See Also

DO ... LOOP, FOR ... NEXT, FUNCTION, SUB

Example

This example uses the EXIT DO statement to break out of a loop when a certain condition is satisfied.

```
dim i as integer
i = 0
do
  i = i + 1
  if i > 10 then exit do
  print i
loop
```

EXP Function

Returns the natural antilogarithm of a value.

Syntax

```
EXP(number)
```

Remarks

This function returns e (the natural logarithm base) raised to the power of *number*.

See Also

LOG

Example

This example raises e to the power of 2 (in other words, it computes the square of e) and prints the result.

```
print exp(2)
```

FINDFIRST Function

Used to find the first of a set of files matching a wildcard filename specification.

Syntax

```
FINDFIRST(name, sr)
```

Remarks

name is the filename specification to search for, and can include DOS wildcard characters * and ?.

sr is a variable of type SEARCHREC that will hold the file information for the file found.

This function is normally used in conjunction with the FINDNEXT function to gather a list of files.

The declaration for the SEARCHREC type is as follows:

```
type searchrec
  FileAttributes as long
  CreationTime as DateTime
  LastAccessTime as DateTime
  LastWriteTime as DateTime
  FileSize as long
  FileSizeHigh as long
  FileName as string*260
  AlternateFileName as string*14
end type
```

This function returns zero if a file matching the specification was found. It returns a nonzero value if no files were found.

See Also

EXISTS, FINDNEXT

Example

This example lists all the files in the “c:\qmwin” directory.

```
dim sr as searchrec, r as integer
r = findfirst("c:\qmwin\*.*", sr)
do while r = 0
  print sr.filename
  r = findnext(sr)
loop
```

FINDNEXT Function

Finds the next of a set of files that was started using the FINDFIRST function.

Syntax

```
FINDNEXT(sr)
```

Remarks

sr is a variable of type SEARCHREC.

This function is always used in conjunction with the FINDFIRST function to search for a list of files.

This function returns zero if another file matching the specification was found. It returns a nonzero value if no more files were found.

See Also

EXISTS, FINDFIRST

Example

See the FINDFIRST function for an example of how to use this function.

FIX Function

Returns the integer or whole number portion for a numeric expression.

Syntax

```
FIX(number)
```

Remarks

FIX and INT operate the same for positive values, however with negative values their operation is slightly different. FIX returns the next-higher integer with negative values, while INT returns the next lower integer with negative values.

See Also

INT

Example

This example shows how the FIX function rounds negative numbers up to the next higher integer. In this example the output will be 3 and 4.

```
print fix(3.7), fix(-4.9)
```

FLUSH Statement

Flushes either or both of the input and output communications buffers.

Syntax

```
FLUSH [INPUT] [OUTPUT]
```

Remarks

The INPUT keyword indicates that the communications input buffer should be flushed.

The OUTPUT keyword indicates that the communications output buffer should be flushed.

See Also

BREAK, HANGUP

Example

This example shows how the FLUSH INPUT statement might be used to get rid of stray input data caused by hanging up the modem.

```
hangup

delay 2

flush input
```

FOR ... NEXT Statement

Executes a sequence of program statements a specified number of times.

Syntax

```
FOR index = initial TO final [STEP step]
  [statements]
NEXT [index]
```

Remarks

index is a numeric variable of type byte, integer, long, or real, and counts the number of times the loop executes.
initial is the initial value assigned to the counter at the beginning of the loop.
final is the ending value against which the loop tests the variable index for each pass.
step is the amount by which the counter is incremented after each pass in a FOR ... NEXT loop. The default value is 1.
When the value of index is higher than the value of final, the loop finishes, and returns control to the statement following the keyword NEXT. If the step value is less than zero, then the index is compared against the final value and the loop is stopped if *index* is less than *final*.

See Also

DO ... LOOP, EXIT, WHILE ... WEND

Example

This example prints “hello” ten times.

```
dim i as integer
for i = 1 to 10
  print "hello "; i
next
```

FORMATDATE Function

Formats a DateTime value into a readable format determined by the picture parameter.

Syntax

```
FORMATDATE (picture as string, dt as datetime)
```

Remarks

The picture parameter is of the form “MM-dd-yy”. Valid items in the picture string are:

Picture	Meaning
d	day of month as digits without a leading zero
dd	day of month as digits with a leading zero
ddd	day of week as a three letter abbreviation
dddd	day of week as a full name
M	month as digits without leading zero
MM	month as digits with a leading zero
MMM	month as a three letter abbreviation
MMMM	month as a full name
y	year as only last digit (if less than 10)
yy	year as last two digits
yyyy	year as full four digits
gg	period/era string

See Also

FORMATTIME,TIME

Example

This example shows how you might use the FormatDate to print the current date as 06/22/95 in the terminal window.

```
dim currentdate as datetime
GetCurrentDateTime(datetime)
print "The date is ";formatdate ("dd/MM/yy", currentdate)
```

FORMATTIME Function

Formats a DateTime value into a readable format determined by the picture parameter.

Syntax

```
FORMATTIME(picture as string,dt as DateTime)
```

Remarks

Valid items in the picture string are:

Picture	Meaning
hh	hour as digits with a leading zero
mm	minutes as digits with a leading zero
ss	seconds as digits with a leading zero
t	time marker string as a single character
tt	time marker as an entire string

See Also

FORMATDATE

Example

This example shows you how to print the current time in the terminal window.

```
dim currentdate as datetime

getcurrentdatetime (currentdate)

Print "The time is ";formattime ("hh:mm:ss tt",currentdate)
```

FREEFILE Function

Returns the next available file number.

Syntax

```
FREEFILE
```

Remarks

You can assign file numbers on the fly by storing the FREEFILE result in a variable and passing the variable to OPEN and CLOSE statements.

Return value

This function returns the first available file number (one that does not refer to a currently open file). If no file number is available, then the return value is –1.

See Also

OPEN

Example

This example opens a file and outputs some test data to it. It doesn't matter how many files are currently open because it uses the FREEFILE function to find a free file number.

```
dim f as integer

f = freefile

open "test.dat" for output as #f

print #f, "this is a test"

close #f
```

FUNCTION Statement

Allows you to define your own subprograms that return a value to the caller.

Syntax

```
FUNCTION name[(arg AS type[, arg AS type]...)] AS type

  ...

  name = expr

  ...

END FUNCTION
```

Remarks

name is the name you assign to the function.

arg is the name of a formal argument to the function. Each argument must have an associated type declaration using the AS keyword.

You may define local variables within your user-defined function. You can use STATIC declarations to preserve the value of individual variables across function calls.

To return a value from the function, use an assignment to the name of the function. The value you assign to the function is remembered until the function ends, at which point it is retrieved and returned to the caller of the function.

Functions cannot be defined within another function definition.

See Also

BYVAL, DECLARE, EXIT, STATIC, SUB

Example

This example declares a function that returns its argument multiplied by two and incremented by one.

```
function f(x as integer) as integer

  f = x*2 + 1

end function


print f(2)

print f(f(7))
```

GET Statement

Reads information from a random-access or binary file into a record variable.

Syntax

```
GET [#]filenumber, [position], variable
```

Remarks

filenumber is the number assigned to an open file.

position is the number of the record in a random access file, or the number of the byte in a binary file. Note that unlike some other languages, the first record or byte in the file is number 1, not number 0. If the position is not specified, then the record is read from the current file position.

variable is the name of the variable that receives the returned data. WARNING: If you are reading a record from a random access file, you must ensure that the variable is of the correct type, otherwise unpredictable results may occur.

See Also

INPUT, OPEN, PUT, TYPE ... END TYPE

Example

This example creates a random access file, writes a record to it from the variable d, and reads it back into the variable z. It then prints out the contents of z to make sure that the operation succeeded.

```
type daterec
  day as integer
  month as integer
  year as integer
end type
dim d as daterec, z as daterec
d.day = 11
d.month = 10
d.year = 1993
open "test.dat" for random as #1 len = len(daterec)
put #1, 1, d
get #1, 1, z
close #1
print z.day, z.month, z.year
```

GETCURRENTDATETIME Function

Gets the current date and time as a DateTime value and places it in the dt variable.

Syntax

```
GETCURRENTDATETIME(dt as DateTime)
```

See Also

FORMATDATETIME, DATETIME

Example

This example shows you how you might use the GetCurrentDateTime function to print the current time in the terminal window.

```
dim currentdate as datetime

getcurrentdatetime (currentdate)

Print "The time is ";formattime ("hh:mm:ss tt",currentdate)
```

GETFIRSTCOUNTRY Function

Fills in a CountryInfo structure with the first country.

Syntax

```
GETFIRSTCOUNTRY (info as CountryInfo) as boolean
```

See Also

GETNEXTCOUNTRY

Example

This example prints a list of country names seen by Windows 95.

```
type countryinfo
  CountryID as integer
  CountryCode as integer
  Name as string*64
end type

dim cinfo as CountryInfo

if GetFirstCountry(cinfo) then
  do
    print cinfo.name
  loop while getnextcountry(cinfo)
end if
```

GETMODEMCOUNT Function

Returns the number of modems available in a system.

Syntax

```
GETMODEMCOUNT as integer
```

See Also

GETMODEMNAME

Example

This example prints the number of modems defined in a system.

```
print "The number of modems configured in this system is "\
   ;getmodemcount
```

GETMODEMNAME Function

Returns the name of modem specified.

Syntax

```
GETMODEMNAME (index as integer) as string
```

Remarks

This function returns the name of the modem specified by the index parameter. The modems are numbered, starting at zero. If GetModemCount indicates there are three installed modems, they are then numbered 0, 1, and 2. The GetModemName function reads these numbers and returns the modem name.

See Also

GETMODEMCOUNT

Example

Here is an example of how you might use this function to print the names of the modems in a defined system.

```
dim count as integer


for count = 0 to getmodemcount -1
  print "one of the modems is a ";getmodemname (count)
next count
```

GETNEXTCOUNTRY Function

Fills in a CountryInfo structure with information on the next country,pass CountryInfo from a previous call to GetFirstCountry.

Syntax

```
GETNEXTCOUNTRY (info as CountryInfo) as boolean
```

See Also

GETFIRSTCOUNTRY

Example

This example prints a list of country names seen by Windows 95.

```
type countryinfo
  CountryID as integer
  CountryCode as integer
  Name as string*64
end type

dim cinfo as CountryInfo

if GetFirstCountry(cinfo) then
  do
    print cinfo.name
  loop while getnextcountry(cinfo)
end if
```

GETPHONEENTRY Function

Retrieves an entry from a phonebook.

Syntax

```
GETPHONEENTRY (index as integer, entry as PhoneEntry) as boolean
```

Remarks

This function retrieves an entry from the phonebook. The phonebook entries are numbered, beginning with zero. If GetPhoneEntryCount returns a value of 10, the entries are numbered 0 through 9. This function returns False if the index is not valid.

See Also

GETPHONEENTRYCOUNT

Example

This is an example of how you might print the systemnames of the entries in the current phonebook

```
dim count as integer

dim entry as phoneentry


for count = 0 to getphoneentrycount - 1

  getphoneentry (count,entry)

  print ";entry.name" is "System #";count+1;

next
```

GETPHONEENTRYCOUNT Function

Returns the number of entries in a phonebook.

Syntax

```
GETPHONEENTRYCOUNT as integer
```

See Also

GETPHONEENTRY, GETMODEMCOUNT

Example

This example prints the number of phone entries in the current phonebook.

```
print "The number of phonebook entries is ";getphonebookentrycount
```

GOSUB ... RETURN Statement

Branches unconditionally to the specified line number or label to execute a subroutine.

Syntax

```
GOSUB {line|label}

    .

label:

    .

RETURN
```

Remarks

line and *label* indicate the line number or label at which the subroutine should begin executing.

The RETURN statement resumes execution at the statement following the original GOSUB statement.

You cannot use GOSUB to branch into or out of a sub-program or user-defined function.

The SUB and FUNCTION constructs provide a much better method of supporting subprograms than GOSUB and RETURN. The use of the GOSUB and RETURN statements is generally not recommended.

See Also

FUNCTION, GOTO, SUB

Example

This example has a very simple subroutine called “printit” that prints the value of i and returns.

```
dim i as integer

i = 5

gosub printit

i = 9

gosub printit

end

printit:

print i

return
```

GOTO Statement

Branches unconditionally to a specified line or label.

Syntax

```
GOTO {line|label}
```

Remarks

Program execution continues at the referenced line or label, or the first executable statement immediately following the line or label.

You cannot use GOTO to branch into or out of a sub-program or user-defined function.

Proponents of structured programming recommend against using GOTO statements wherever IF statements and DO loops can accomplish the same task, on the grounds that structured code is more efficient and easier to maintain.

See Also

FUNCTION, GOSUB, SUB

Example

This example demonstrates a simple loop structure using a GOTO statement.

```
dim i as integer

i = 0

again:

i = i + 1

if i < 10 then goto again
```

HANGUP Statement

Hangs up the modem by sending the modem hangup string, without asking for confirmation.

Syntax

```
HANGUP
```

Remarks

This command tells the system to hang up. The hangup command is sent through Windows 95.

See Also

BREAK, CARRIER

Example

This example sends a command to log off an on-line service, waits five seconds, then hangs up the modem.

```
send "bye"

delay 5

hangup
```

HEX Function

Converts an integer or long integer expression to a hexadecimal (base 16) value.

Syntax

```
HEX(number)
```

Remarks

number is the numeric variable or expression that HEX converts. The value may be any numeric type — it will be converted to an integer or a long integer. Fractional values are ignored.

See Also

OCT

Example

This example prints the hexadecimal representation of the number 42.

```
print hex(42)
```

HOSTECHO Function

Used to turn on and off character echo conventions that are used with the host mode.

Syntax

```
HOSTECHO [ON | OFF]
```

Remarks

This command modifies the operation of *QmodemPro* in the following ways:

Incoming data is not displayed on the screen, written to the capture file, or sent to the printer. It is only sent to the script program for processing.

Data that is output to the communications port is automatically echoed to the terminal screen.

When using the SEND command to output data to the communications port, a carriage return/line feed pair is sent at the end of the line if no semicolon is present (this is in contrast to the standard behavior of sending just a carriage return).

See Also

DUPLEX

Example

This example shows what will happen when HOSTECHO is turned on. For the best example of how this command works, see the host mode script HOST.QSC.

```
dim name as string

hostecho on

send "What is your first name? ";
```

IF Statement

Makes a decision regarding program flow, based on the result returned by an expression.

Syntax

```
IF expression THEN then-part ELSE else-part

or

IF expression1 THEN
   statements-1
[ELSEIF expression2 THEN
   statements-2]
[ELSE
   statements-n]
END IF
```

Remarks

expression is an expression that yields a result of true (nonzero) or false (zero).

then-part is a set of statements to be executed if *expression* is true.

else-part is a set of statements that will be executed if *expression* is false.

ELSE and ELSEIF are optional. ELSE allows you to execute a set of statements if *expression* is false.

ELSEIF allows you to test for several different conditions within a single IF statement. In the construct above, *expression2* would only be tested if *expression1* was false.

See Also

DO ... LOOP, SELECT CASE

Example

This example demonstrates both types of IF statement syntax.

```
dim i as integer
i = 5
if i = 7 then print "is seven" else print "not seven"
if i = 5 then
  print "i is 5"
elseif i = 9 then
  print "i is 9"
else
  print "i is neither 5 nor 9"
end if
```

IMP Operator

IMP performs a bitwise implication operation between its operands.

Syntax

```
op1 IMP op2
```

Remarks

op1 and *op2* are logical expressions or numeric values of any integral type (byte, integer, or long integer).

This is the logical truth table for the IMP operator:

a	b	a IMP b
0	0	1
0	1	1
1	0	0
1	1	1

See Also

AND, EQV, NOT, OR Operator , XOR

Example

This example prints out the truth table above.

```
dim a as integer, b as integer
print "a", "b", "a IMP b"
for a = 0 to 1
  for b = 0 to 1
print a, b, a IMP b
  next
next
```

INCDATETIME Function

Increments a DateTime value.

Syntax

```
INCDATETIME(dt1 as DateTime, dt2 as DateTime, days as integer, seconds as integer)
```

Remarks

Increments the DateTime value dt1 by the number of seconds and days specified the last two parameters, and places the results in the dt2 variable.

Example

This example prints the date seven days from today.

```
dim currentdate as datetime
dimfuturedate as datetime

getcurrentdatetime (currentdate)
incdatetime (currentdate,futuredate,7,0)
print "In 7 days, the date will be";formatdate \
   ("ddd, MMMM dd, yyyy",futuredate)
```

INKEY Function

Reads a character from the keyboard, communications port, or a file.

Syntax

```
INKEY(filenumber)
```

Remarks

If there is no character ready when reading from the keyboard or communications port, this function returns an empty string.

When reading from a file or communications port, or if there is an ordinary character waiting from the keyboard, this function returns a single-character string containing the character.

When reading from the keyboard, the following strings are returned if the corresponding special key is pressed:

```
"Tab", "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9", "F10", "F11", "F12",
"Backspace", "Enter", "Scroll Lock", "Pause", "Gray Insert", "Gray Delete", "Gray
Home", "Gray End", "Gray Pageup", "Gray Pagedown", "Gray Up", "Gray Down", "Gray
Left", "Gray Right", "Numlock", "Pad /", "Pad *", "Pad -", "Pad +", "Pad Enter",
"Pad .", "Pad 0", "Pad 1", "Pad 2", "Pad 3", "Pad 4", "Pad 5", "Pad 6", "Pad 7", "Pad
8", "Pad 9", "Pad Delete", "Pad Insert", "Pad End", "Pad Down", "Pad Pagedown", "Pad
Left", "Pad Clear", "Pad Right", "Pad Home", "Pad Up", "Pad Pageup"
```

See Also

INPUT

Example

This example reads a single keypress from the keyboard and prints the resulting string.

```
dim c as string
do
  c = inkey
loop until c <> ""
print "key pressed: "; c
```

INPUT Statement

Captures a line of data from the keyboard, communications port, or a file, and places the data into a string variable.

Syntax

```
INPUT [#filenum,] variable
```

Remarks

filenum is the file number to use.

variable is the variable name into which you want to place the captured data.

See Also

GET, INKEY

Example

This example simply reads a line of input from the user and prints it back out again.

```
dim a as string

input a

print a
```

INSTR Function

Searches for the first occurrence of a string of characters within a specified string.

Syntax

```
INSTR([start,] string1, string2)
```

Remarks

start is the position at which to begin the search. This parameter is optional — the default is to start searching at position 1.

string1 is the string to search.

string2 is the text to search for within the string.

INSTR returns the position of the first occurrence of the specified text. If INSTR does not find the specified string, it returns a value of 0.

See Also

LEFT, MID, RIGHT

Example

This example looks for the string “Windows 95” within the string “*QmodemPro for Windows 95*”.

```
dim a as string

a = "QmodemPro for Windows 95"

print instr(a, "Windows 95")
```

INT Function

Returns the integer or whole number portion of a numeric expression.

Syntax

```
INT(expr)
```

Remarks

FIX and INT operate the same for positive values, however with negative values their operation is slightly different. FIX returns the next-higher integer with negative values, while INT returns the next lower integer with negative values.

See Also

FIX

Example

This example shows how the INT function rounds negative numbers down to the next lower integer. In this example the output will be 3 and 5.

```
print int(3.7), int(-4.9)
```

INTEGER Type

Used to declare a variable that can handle integer numbers.

Remarks

Variables of integer type can hold values that range from 2147483648 to 214783647.

See Also

BYTE, DIM, LONG, REAL, SHORT, TYPE

Example

This example declares a variable of type INTEGER and assigns a value to it.

```
dim i as integer

i = 500000

print i
```

KILL Statement

Deletes a file from disk.

Syntax

```
KILL filename
```

Remarks

filename is the name of the file to delete. DOS wildcards * and ? are not supported.

You cannot delete an open file.

This statement is identical to the script DEL command.

See Also

DEL, RMDIR

Example

This example removes the file “test.dat” from the current directory.

```
kill "test.dat"
```

LASTCONNECTPASSWORD Function

This function makes the last password entry available.

Syntax

```
LASTCONNECTPASSWORD
```

Remarks

The password returned corresponds to the dialing directory entry that was used for the most recent connection.

See Also

DIAL, LASTCONNECTUSERID

Example

```
waitfor "password?"
send lastconnectpassword
```

LASTCONNECTUSERID Function

This function returns the current user id.

Syntax

```
LASTCONNECTUSERID
```

Remarks

The user id returned corresponds to the dialing directory entry that was used for the most recent connection.

See Also

DIAL, LASTCONNECTPASSWORD

Example

```
waitfor "name:"

send lastconnectuserid
```

LCASE Function

Returns a copy of a string with all upper case characters converted to lower case.

Syntax

```
LCASE(string)
```

Remarks

string can be any string expression.

This operation is useful for making case insensitive comparisons of text. The UCASE function operates similarly, but converts the specified text to upper case.

See Also

UCASE

Example

This example demonstrates using the LCASE function to print a lower case version of a string.

```
dim a as string

a = "QmodemPro for Windows 95"

print lcase(a)
```

LEFT Function

Returns a string consisting of a specified number of characters starting at the left (beginning) of a string.

Syntax

```
LEFT(string, num)
```

Remarks

string is any string expression.

num is any number in the range 0 through 32767.

If there are fewer than *num* characters in *string*, the entire string will be returned without any padding.

See Also

INSTR, MID, RIGHT

Example

This example uses the LEFT function to print the first word in the string “*QmodemPro for Windows 95*”.

```
dim a as string

a = "QmodemPro for Windows 95"

print left(a, 9)
```

LEN Function

Returns the number of characters in a string, or returns the size in bytes of a user defined type.

Syntax

```
LEN(string)

or

LEN(typename)
```

Remarks

string is any string expression. Used in this way the function will return the number of characters in the string.

typename is the name of any user defined type. Used in this way the function will return the number of bytes taken up by a variable of the indicated type. This is useful when used with the LEN = clause in the OPEN statement.

See Also

INSTR, LEFT, MID, OPEN, RIGHT

Example

This example prints the length of the string "*QmodemPro for Windows 95*". See the OPEN script command for an example of how to use LEN with OPEN.

```
dim a as string

a = "QmodemPro for Windows 95"

print len(a)
```

LET Statement

Assigns the value of an expression to a variable.

Syntax

```
[LET] var = expr
```

Remarks

The LET keyword is an assignment statement — it allows you to assign values to variables.

You must declare a variable using DIM before you can assign a value to it.

The keyword itself is optional, so statements such as

```
LET tries = 1
```

is functionally identical to

```
tries = 1
```

The LET keyword is usually omitted in all cases.

See Also

DIM, TYPE

Example

This example shows how to declare a variable, assign a value to it, and print its value on the screen.

```
dim i as integer

i = 5

print i
```

LOADPHN Statement

Loads a new phonebook.

Syntax

```
LOADPHN filename
```

Remarks

This command loads a new phone book just like the phonebook's **File/Open** command would.

See Also

DIAL

Example

This example shows how to load a new phone book and dial its first entry.

```
loadphn "test.phn"

dial entry 0
```

LOADTRANSLATETABLE Function

Loads a translation table.

Syntax

```
LOADTRANSLATETABLE filename
```

Remarks

This command loads a translation table just like the the Emulation property sheet would.

Example

This example shows how print out verification of a loaded translation table “NewTrans”.

```
If LoadTranslateTable ("NewTrans") then

  print "New Translation Table Loaded."

else

  print "Translation Table failed to load."

End IF
```

LOC Function

Returns the current position of a pointer in an open file, indicating the point where the next read or write operation will take place.

Syntax

```
LOC(filenumber)
```

Remarks

filenumber is the number assigned to the open file.

If the file is opened for sequential or binary access, LOC returns the current file position in bytes.

If the file is opened for random access, LOC returns the current record number based on the record size specified in the LEN = clause in the OPEN statement.

See Also

GET, OPEN, PUT, SEEK

Example

This example writes an integer to a file and reports the new file position. In this case the file position after writing will be 2.

```
dim i as integer

i = 5

open "test.dat" for random as #1 len = len(integer)

put #1, 1, i

print loc(1)

close #1
```

LOCATE Statement

Positions the cursor on the screen.

Syntax

```
LOCATE row, column
```

Remarks

row is the number of the row at which the cursor is positioned. Rows are numbered 1 through the maximum number of rows on the screen.

column is the number of the column at which the cursor is positioned. Columns are numbered 1 through the number of columns on the screen.

See Also

CSRLIN, POS

Example

This example places the cursor on row 5, column 5, and writes a string indicating that fact.

```
locate 5, 5

print "this is at position 5, 5"
```

LOF Function

Returns the number of records in an open file.

Syntax

```
LOF(filenumber)
```

Remarks

filenumber is the number under which the file was opened.

If the file is opened for sequential or binary access, LOF returns the size of the file in bytes.

If the file is opened for random access, LOF returns the size of the file based on the record size specified in the LEN = clause in the OPEN statement.

See Also

LOC, OPEN

Example

This example opens a file for sequential access and outputs the length of the file.

```
open "test.dat" for input as #1

print lof(1)

close #1
```

LOG Function

Returns the natural logarithm of a number.

Syntax

```
LOG(expression)
```

Remarks

expression is any numeric expression.

A natural logarithm is the power to which the constant e (about 2.718282) must be raised to obtain a given number. This should not be confused with common logarithms, which are based on 10 rather than e.

If you try to take the logarithm of zero or a negative number, the ERR_MATH error will be generated. You can catch this error with the CATCH statement.

See Also

CATCH, EXP

Example

This example prints the logarithm of the number 6.

```
print log(6)
```

LOGFILE Statement

Turns on or off the log file.

Syntax

```
LOGFILE {ON | OFF}
```

Remarks

This command turns on or off the log file just like the **File/Log Toggle** command from the main terminal menu. The log file name is defined in **Options/Files/File Definitions**.

See Also

CAPTURE

Example

This example turns on the log file, uploads a file, then turns it off again.

```
logfile on

call upload "c:\qmwin\test.dat", Zmodem

logfile off
```

LONG Type

Used to declare a variable that can handle integer numbers larger than those handled by the SHORT type.

Remarks

Variables of the long type can hold values that range from 2147483648 to 2147483647.

See Also

BYTE, DIM, INTEGER, REAL, TYPE

Example

This example declares a variable of type long, assigns a number to it, and prints out the number.

```
dim i as long

i = 500000

print i
```

LOOP Statement

Used to mark the end of a DO ... LOOP statement. See the DO statement for more information.

LTRIM Function

Trims leading spaces (spaces on the left) from a string expression.

Syntax

```
LTRIM(string)
```

Remarks

string is any string expression.

See Also

RTRIM

Example

This example trims the leading spaces off the given string and prints the results within angle brackets.

```
dim a as string

a = "  QmodemPro for Windows 95  "

print ">"; ltrim(a); "<"
```

MAXIMIZE Statement

Causes *QmodemPro for Windows 95* to maximize its application window on the Windows 95 desktop.

Syntax

```
MAXIMIZE
```

Remarks

The *QmodemPro for Windows 95* application window is maximized and brought to the front of the Windows 95 desktop.

See Also

ACTIVATE, MINIMIZE, MOVE, SIZE

Example

This example causes the *QmodemPro for Windows 95* application to maximize its window.

```
MAXIMIZE
```

MID Function

Retrieves a substring from an arbitrary position in another string.

Syntax

```
MID(string, start[, num])
```

Remarks

string is any string expression.

start is the position of the first character to copy.

num (optional) is the number of characters to return from within the string expression. If the number is omitted, the remainder of the string is returned.

If there are not enough characters in the string to supply enough characters in the result, a shorter string than that asked for will be returned.

See Also

LEFT, RIGHT

Example

This example prints some characters from the middle of a given string.

```
dim a as string

a = "QmodemPro for Windows 95"

print mid(a, 7, 11)
```

MINIMIZE Statement

Causes *QmodemPro for Windows 95* to minimize its application window on the Windows 95 desktop. Minimizing an application causes it to appear only on the taskbar.

Syntax

```
MINIMIZE
```

Remarks

The *QmodemPro for Windows 95* application window is minimized. Any other Windows 95 opened by *QmodemPro for Windows 95* (phonebook, GIF viewer, editor, and so on) will disappear until the application is restored.

See Also

ACTIVATE, MAXIMIZE, MOVE, SIZE

Example

This example causes the *QmodemPro for Windows 95* application to minimize itself.

```
MINIMIZE
```

MKDIR Statement

Creates a directory on the disk.

Syntax

```
MKDIR directory
```

Remarks

directory is the name of the directory to create.

When creating a new directory, each of the directories leading up to the last directory name must already exist. If you try to create the directory “c:\abc\def” and the directory “c:\abc” does not already exist, you will get an ERR_PATH error.

See Also

CHDIR, CURDIR, RMDIR

Example

This example makes a new directory called “test” under the “c:\qmwin” directory.

```
mkdir "c:\qmwin\test"
```

MOD Operator

Returns the remainder of a division between two integers.

Syntax

```
op1 MOD op2
```

Remarks

op1 and *op2* can be of any integer numeric type (byte, integer, long).

If *op2* is zero the ERR_MATH error will be generated.

Example

This example prints 144, which is the remainder after 400 is divided by 256.

```
print 400 mod 256
```

MOUSECLICK Statement

Used to simulate a mouse click for the RIPscrip emulation.

Syntax

```
MOUSECLICK x, y
```

Remarks

This statement simulates a mouse click on an area of the screen when using the RIPscrip emulation.   The x and y parameters are the horizontal and vertical coordinates of the mouse click.   This command is generated by Quicklearn when using the RIPscrip emulation.

If the RIPscrip emulation is not active, this command has no effect.

See Also

EMULATION

Example

This example simulates a mouse click at position 10, 10 on the screen.

```
mouseclick 10, 10
```

MOVE Statement

Used to move the *QmodemPro for Windows 95* application window on the Windows 95 desktop.

Syntax

```
MOVE x, y
```

Remarks

x is the horizontal coordinate of the new position of the window. X coordinate values range from 0 to the width of your screen minus one.

y is the vertical coordinate of the new position of the window. Y coordinate values range from 0 to the height of your screen minus one.

If you are using standard VGA with a 640x480 screen then the X values can range from 0 through 639 and Y values can range from 0 through 479.

See Also

ACTIVATE, MAXIMIZE, MINIMIZE, SIZE

Example

This example moves the *QmodemPro for Windows 95* application window so its upper left hand corner is at position (300, 200).

```
MOVE 300, 200
```

MSGBOX Statement

Used to pop up a simple message box with a message and an OK button.

Syntax

```
MSGBOX string
```

Remarks

This statement pops up a simple message box with an OK button and waits for the user to press the OK button before continuing.

See Also

DIALOG, DIALOGBOX

Example

This example pops up a simple message box on the screen.

```
msgbox "Hello world!"
```

NAME Statement

Renames or moves a file.

Syntax

```
NAME oldfilespec AS newfilespec
```

Remarks

oldfilespec is the current file name.

newfilespec is the new file name.

This command works in a similar way to the DOS RENAME command, with the added ability to move a file, if the new filename includes a path.

You cannot use the NAME command to rename a directory.

If an error occurs during the rename operation, the ERR_FILERENAME error will be generated.

See Also

DEL, FINDFIRST, FINDNEXT, KILL

Example

This example renames the file “test.fil” to the file “test.dat”.

```
name "test.fil" as "test.dat"
```

NEXT Statement

Marks the end of a FOR ... NEXT loop. See the FOR command for more information and examples.

NOT Operator

Performs a bitwise logical NOT operation.

Syntax

```
NOT op
```

Remarks

This operator takes the binary representation of *op* and reverses the status of each bit in the representation.

When using NOT in an IF statement, the value FALSE (zero) will be converted to TRUE (–1) and vice versa.

See Also

AND, EQV, IF Statement , IMP, OR Operator , XOR

Example

This example prints out the result of using NOT on various values.

```
print not true

print not false

print not 1
```

OCT Function

Converts an integer expression to an octal (base 8) representation.

Syntax

```
OCT(expr)
```

Remarks

The expression passed to this function will be converted to a long integer before being converted to an octal representation.

See Also

HEX, STR, VAL

Example

This example prints the octal representation of the number 42.

```
print oct(42)
```

OPEN Statement

Opens a file so the program can perform input and output operations.

Syntax

```
OPEN file FOR accessmode AS [#]filenum [LEN = reclen]
```

Remarks

file is the name of the file to open.

accessmode specifies the way the program will write to or read from file: INPUT, OUTPUT, APPEND, RANDOM and BINARY.

filenum is the file number to assign to the file.

reclen is the length of a record in a sequential or random-access file, in the range 1 through 32767.

See Also

CLOSE, FREEFILE

Example

This example opens a file for sequential output and writes a string of test data to it.

```
open "test.dat" for output as #1

print #1, "this is a test"

close #1
```

OPENSERIALPORT Function

Opens a serial port.

Syntax

OPENSERIALPORT(portname as string)

Remarks

This function opens a serial port. The valid names are generally either “COM1” or “COM2”, but you may use other names, depending on the hardware and drivers you have installed. When using this function, do not place a colon (:) after the port name.

If the port is successfully opened, this function returns True.

See Also

OPENTCPIPPORT, CLOSEPORT

Example

Here is an example of how you might open COM1, send a dial command, and then close the port. This example includes commands to print a message to the screen when the port is opened and closed.

```
if openserialport ("COM1") then print "COM 1 opened."

SEND "ATDT 1-805-873-2400 ^M"

Closeport

print "port closed."
```

OPENTCPIPPORT Function

Opens a connection using TCP/IP protocol.

Syntax

OPENTCPIPPORT(host as string, port as integer, telnet as boolean)

Remarks

This function opens a TCP connection to a specific port on another host using the TCP/IP protocol. The host parameter is either the name of the host (rs.internic.net, for example} or its IP address (such as 198.41.0.6).

The port paramter is optional, and defaults to 23, the Telnet port.

The Telnet parameter is also optional, and defaults to True, which means that Telnet protocol will be used during the connection. Passing False as this parameter will open the port in “raw” mode, where no Telnet-specific processing will take place.

If the port is successfully opened, this function returns True.

See Also

OpenSerialPort, ClosePort

Example

Here is an example of how this function could be used to open a TCP/IP port and connect to MSI HQ BBS.

```
if OpenTCPIPPort("bbs.mustnag.com") then print "Connected to Mustang Software BBS"
```

OR Operator

OR performs a bitwise logical or operation between its operands.

Syntax

```
op1 OR op2
```

Remarks

op1 and *op2* are logical expressions or numeric values of any integral type (byte, integer, or long integer).

This is the logical truth table for the OR operator:

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

See Also

AND, EQV, IMP, NOT, XOR

Example

```
print "5 or 6 is 7: "; 5 or 6
```

PAUSE Statement

Used to suspend script execution for a certain time interval.

Syntax

```
PAUSE time
```

Remarks

time is the amount of time to suspend execution, expressed in seconds. If you want to delay for less than a second, use ordinary decimal notation.

This command is identical to the script DELAY command.

See Also

DELAY, WAITFOR, WHEN QUIET, WHEN TIME

Example

This example sends “atz” to initialize the modem, pauses for half a second, and sends a command to dial the modem.

```
send "atz"

pause 0.5

send "atdt5551212"
```

PLAY Statement

Used to play a .wav file using the Windows 95 multimedia services.

Syntax

```
PLAY filename
```

Remarks

If Windows 95 cannot find the named .wav file to play, it will sound a default beep (which may be another .wav file). The .wav file is played in “asynchronous” mode which means that Windows 95 will not wait for the sound to finish, and will continue execution with the next script command immediately.

See Also

BEEP, SOUND

Example

This example plays the sound in the file “soundfil.wav”.

```
PLAY "soundfil.wav"
```

POS Function

Reports the current horizontal coordinate position (column number) of the cursor.

Syntax

```
POS
```

Remarks

The return value of this function ranges from 1 to the maximum number of columns on the screen (either 80 or 132).

See Also

CSRLIN, LOCATE

Example

This example prints the current horizontal cursor position.

```
print pos
```

PRINT Statement

Outputs data to the display, communications port, or a file.

Syntax

```
PRINT [#filenum, ][expression[[;|,]expression ...]][;|,]
```

Remarks

expression is an expression of any type except a user defined type.

Multiple items can be separated with semicolons (to display the next item immediately after the previous item), or commas (to display the next item in the next 14-character wide “print zone”). A semicolon at the end of the line will suppress the automatic carriage return and line feed that occurs after printing.

The PRINT statement by itself moves the cursor to the beginning of the next line.

See Also

LOCATE, TAB

Example

This example shows the famous “Hello world” program in its entirety.

```
print "Hello, world!"
```

PRINTER Statement

Turns the printer toggle on or off.

Syntax

```
PRINTER onoff
```

Remarks

This command turns the printer toggle on and off.   When on, all incoming data is echoed to the printer as well as the screen.   The destination of the printer data is controlled by the “Print file” option in Options/Files/File definitions.

See Also

CAPTURE

Example

This example shows how to turn the printer on, wait for some specified data from the port, then turn the printer off again.

```
print on

send

waitfor "Ready"

print off
```

PUT Statement

Writes data to a random access or binary file, from record variables defined by a TYPE ... END TYPE statement.

Syntax

```
PUT [#]filenumber, [position], variable
```

Remarks

filenumber is the file number assigned to the open file.

position is the number of the random access file record, or binary file byte to begin writing. Note that unlike some other languages, the first record or byte in the file is number 1, not number 0. If this position is not specified, then the record is written to the current file position.

variable is the name of the variable that contains the data to place in the file.

See Also

GET, INPUT, OPEN, TYPE

Example

This example creates a random access file, writes a record to it from the variable d, and reads it back into the variable z. It then prints out the contents of z to make sure that the operation succeeded.

```
type daterec
  day as integer
  month as integer
  year as integer
end type
dim d as daterec, z as daterec
d.day = 11
d.month = 10
d.year = 1993
open "test.dat" for random as #1 len = len(daterec)
put #1, 1, d
get #1, 1, z
close #1
print z.day, z.month, z.year
```

REAL Type

Used to declare a variable that can handle real (floating-point) numbers.

Remarks

The range of REAL numbers is 3.4e38 to 3.4e38.

See Also

BYTE, DIM, INTEGER, LONG, STRING, TYPE

Example

This example declares a variable of type real, assigns a value to it, and prints out the value.

```
dim x as real

x = 1.5

print x
```

RECEIVE Statement

Captures a line of data from the communications port (by default) and places the data into a string variable.

Syntax

```
RECEIVE [#filenum,] var
```

Remarks

This statement is identical to the INPUT statement except that if a file number is not specified, the data is input from the communications port instead of the keyboard.

See Also

INPUT

Example

This example reads a line of input from the communications port.

```
dim s as string

receive s
```

RECEIVEFILE Function

Used to receive a file or files from a remote computer.

Syntax

```
RECEIVEFILE(filename, protocol)
```

Remarks

This function initiates a file transfer to receive files from a remote computer. At the time this command is executed, the remote computer must already have started the file transfer. If you are connecting to a bulletin board system (BBS) then your script should already have sent the command to the BBS that will start the file transfer.

protocol is one of the following predefined constants:

ASCII, XMODEM, XMODEMCRC, XMODEM1K, XMODEM1KG, YMODEM, YMODEMG, ZMODEM, KERMIT

For the first five protocols, the *filename* parameter must specify an actual file name in which to place the received file. The ASCII and Xmodem variant protocols do not supply a filename, so one must be supplied in the DOWNLOAD function.

For the last four protocols, the *filename* parameter should be the name of a directory in which to place the received files. If this parameter is an empty string (“”) then the download directory specified in **Options/Files/Path Definitions** will be used.

The RECEIVEFILE function returns zero if the file transfer was successful. If the transfer was unsuccessful, RECEIVEFILE returns the error code describing the error. For a list of error codes see Chapter 4.

This function is identical to the DOWNLOAD function.

See Also

DOWNLOAD, SENDFILE, UPLOAD

Example

This example receives a file using the Zmodem protocol, assuming the remote computer has already started the transfer.

```
if receivefile("", Zmodem) = 0 then

  print "file transfer ok!"

end if
```

REM Statement

Allows you to include a comment, or non-executing text for your own reference, in a program.

Syntax

```
REM comment
```

or

```
' comment
```

or

```
// comment
```

Remarks

The apostrophe ‘ character is a synonym for the REM statement.

Either variation of the statement will cause the compiler to ignore all text on a line following the ‘ or REM statement.

When adding a comment to a line of executable source code, the REM must be preceded by a colon (:). A colon is not required if you are using the apostrophe.

Comments do not affect the size or execution speed of the compiled script.

Example

This example shows two comment lines and one line that is not a comment.

```
rem This is a comment.

' This is another comment, like the above line.

print "This is not a comment."
```

REMOVEPHONEENTRY Statement

Removes a phonebook entry.

Syntax

```
RemovePhoneEntry(entry as PhoneEntry)
```

Remarks

This function removes the phonebook entry specified by the entry parameter. The function returns True if the entry was removed successfully

See Also

ADDPHONEENTRY, UPDATEPHONEENTRY

Example

This is an example of how you might remove all occurences of “MUSTANG” from your currently open phonebook.

```
dim count as integer
dim entry as phoneentry
count = 0
do while count < getphoneentrycount
  print "Number of phone entries is ";getphoneentrycount
  getphoneenrty (count,entry)
  print "Removing Entry #";count;" : ";entry.name
  if instr (ucase(entry.name), "MUSTANG") then
    if removephoneentry (entry) then
      print "Removed Successfully"
    else
      print "Failed Removal"
    end if
  else
    print "Didn't find Mustang in ";entry.name
    count = count + 1
  end if
loop
```

RESET Statement

Description

Closes all open files.

Syntax

```
RESET
```

Remarks

The RESET statement works the same way as the CLOSE statement without parameters. The functions CLOSE, END, STOP, and SYSTEM also close any open files.

See Also

CLOSE, END, STOP, SYSTEM

Example

This example opens a file for input, then closes all open files.

```
open "test.dat" for input as #1

reset
```

RESETEMULATION Statement

Resets the current emulation.

Syntax

```
RESETEMULATION
```

Remarks

This command resets the operation of the currently selected emulation.   It is identical to the Reset Emulation command on the Terminal menu.

See Also

EMULATION

Example

This example shows how the RESETEMULATION command might be used.

```
hangup

resetemulation
```

RETURN Statement

Returns from a subroutine started with the GOSUB statement. See the GOSUB statement for more information and examples.

RIGHT Function

Returns a string consisting of a specified number of characters starting at the right (end) of a string.

Syntax

```
RIGHT(string, num)
```

Remarks

string is any string expression.

num is any number in the range 0 through 32767.

If there are fewer than num characters in the string, the entire string is returned even though it is shorter than num.

See Also

INSTR, LEFT, MID

Example

This example shows how the RIGHT function can be used to extract the rightmost characters from another string.

```
dim a as string

a = "QmodemPro for Windows 95"

print right(a, 7)
```

RMDIR Statement

Removes a directory from a disk.

Syntax

```
RMDIR directory
```

Remarks

directory is the name of the directory to remove.

You cannot remove a directory if it still contains files.

If the directory cannot be removed, the ERR_PATH error is generated. Use the CATCH statement to respond to this error.

See Also

CHDIR, KILL, MKDIR

Example

This example removes the directory “c:\qmwin\test”, assuming it is empty.

```
rmdir "c:\qmwin\test"
```

RND Function

Returns a pseudo-random real number between greater than or equal to 0 and less than 1.

Syntax

```
RND[(number)]
```

Remarks

If *number* is omitted or greater than zero, the next random number in sequence is generated.

If *number* is less than zero, RND returns the same random number every time, depending on the value of expr.

If *number* is zero, the last random number generated is returned.

Example

This example shows the three ways of using the RND function.

```
print rnd(-0.5)

print rnd

print rnd(0)
```

RTRIM Function

Trims trailing spaces (spaces on the right) from a string expression.

Syntax

```
RTRIM(string)
```

Remarks

string is any string expression. Any trailing spaces are removed from the string and the result is returned.

See Also

LTRIM

Example

This example shows how the RTRIM function can be used to trim trailing spaces from a string.

```
dim a as string

a = "  QmodemPro for Windows 95  "

print ">"; rtrim(a); "<"
```

SCREEN Function

Returns the character or color attribute at a particular screen position.

Syntax

```
SCREEN(x, y [, attr])
```

Remarks

Returns the character or attribute (color) at a particular screen position. The screen position is given by the *x* and *y* parameters and ranges from 1 to the with or height of the terminal screen.

If the *attr* parameter is omitted or is zero, this function returns the ASCII value of the character on the screen at the given position. If the attr parameter is one, the return value is the attribute of the character on the screen at the given position.

The attribute returned is defined by the following formula:

```
b * 16 + f
```

where b is the background color and f is the foreground color, as listed under the COLOR statement.

See Also

COLOR

Example

This example clears the screen, writes “Hi” in gray on blue, and prints the ASCII value of “H” followed by the color value 71.

```
cls

color 7, 4

print "Hi"

print screen(1, 1), screen(2, 1, 1)
```

SCROLLBACK Statement

Turns scrollback mode on or off.

Syntax

```
SCROLLBACK onoff
```

Remarks

This command turns the scrollback mode on or off.   The operation of the script is not affected while in scrollback mode (ie. the script continues to execute).

See Also

CAPTURE, SCROLLBACKRECORD

Example

This example shows how to capture some data, then automatically go to scrollback mode to review it.

```
send

waitfor "Command"

scrollback on
```

SCROLLBACKRECORD Function

Used to get or set the current state of the scrollback record toggle.

Syntax

```
SCROLLBACKRECORD
```

```
or
```

```
SCROLLBACKRECORD(onoff)
```

Remarks

There are two forms to the SCROLLBACKRECORD function. The first form takes no arguments and simply returns the current setting. The second form sets the state of the toggle and returns the previous state.

In both the parameter and the return value, a nonzero value means scrollback record mode is turned on. A zero value means scrollback record mode is turned off.

When scrollback record mode is on, incoming data is captured to the scrollback buffer for later review. When scrollback record mode is turned off, data is not copied to the scrollback buffer.

See Also

CAPTURE, SCROLLBACK

Example

This example shows how the SCROLLBACKRECORD function might be used to automatically capture some data.

```
scrollbackrecord on

send

waitfor "End of data"

scrollbackrecord off
```

SEEK Statement

Sets the current position in an open file for the next I/O operation.

Syntax

```
SEEK [#]filenumber, position
```

Remarks

filenumber is the number allocated to the open file.

position is the number of the record or byte to be accessed. Note that unlike many programming languages, the records and bytes in a file are numbered starting at 1 rather than at 0.

See Also

GET, LOC, OPEN, PUT

Example

This example reads from “c:\autoexec.bat” the string starting at offset 50.

```
dim s as string

open "c:\autoexec.bat" for input as #1

seek #1, 50

input #1, s

close #1
```

SELECT CASE Statement

Allows a number of expressions to be compared, executing statements based on the results of the comparison.

Syntax

```
SELECT CASE expression
  CASE case-condition[, case-condition ...]
    [statements]
  [CASE case-condition[, case-condition ...]
    [statements]]
  [CASE ELSE
    [statements]]
END SELECT
```

Remarks

expression is a numeric or string expression.

case-condition is a condition in one of the following forms:

```
expression
expression TO expression
IS relational-operator expression
```

If the *expression* matches the *case-condition*, the statements matching the corresponding CASE are executed. Control then passes to the first statement following END SELECT.

If none of the *case-conditions* matches and there is a CASE ELSE clause, the statements within the CASE ELSE clause are executed.

See Also

IF Statement

Example

This example tests the numbers 0 through 11 with a SELECT CASE statement and prints the results.

```
dim i as integer
for i = 0 to 11
  select case i
    case 1
      print "i is 1"
    case 2 to 9
      print "i is between 2 and 9 inclusive"
    case is >= 10
      print "i is greater than or equal to 10"
    case else
      print "i must be zero or negative"
  end select
next i
```

SEND Statement

Outputs data to the communications port.

Syntax

```
SEND [expression[[;|,]expression ...]][;|,]
```

Remarks

expression is an expression of any type except a user defined type.

Multiple items can be separated with semicolons (to display the next item immediately after the previous item), or commas (to display the next item in the next 14-character wide "print zone").

A semicolon at the end of the line will suppress the automatic carriage return after the transmitted data.

The SEND statement by itself sends only a carriage return.

See Also

LASTCONNECTPASSWORD, LASTCONNECTUSERID, PRINT

Example

This example shows how the SEND command may be used to send various data to the modem.

```
send "bob"       ' sends "bob" followed by Enter

send "g";        ' sends "g" without an Enter

send lastconnectuserid

                 ' sends the UserId corresponding

                 ' to the phone directory entry

                 ' you are currently connected to
```

SENDFILE Function

Used to send files to a remote computer.

Syntax

```
SENDFILE(filename, protocol)
```

Remarks

This function initiates a file transfer to send files to a remote computer. At the time this command is executed, the remote computer must already have started the file transfer. If you are connecting to a bulletin board system (BBS) then your script should already have sent the command to the BBS that will start the file transfer.

The protocol is one of the following predefined constants:

ASCII, XMODEM, XMODEMCRC, XMODEM1K, XMODEM1KG, YMODEM, YMODEMG, ZMODEM, KERMIT

The first five protocols require a single file name in the filename parameter. This file name indicates the file to send.

The last four protocols can accept more than just one file in the filename parameter — separate multiple filenames with spaces.

The SENDFILE function returns zero if the file transfer was successful. If the transfer was unsuccessful, SENDFILE returns the error code describing the error. For a list of error codes see Chapter 4.

This function is identical to the UPLOAD function.

See Also

DOWNLOAD, RECEIVEFILE, UPLOAD

Example

This example shows how to send a file using the Zmodem protocol, assuming the remote computer has been told to accept a file transfer.

```
if sendfile("c:\qmwin\test.dat", Zmodem) = 0 then
  print "file transfer ok!"
end if
```

SETCOMM Statement

Sets the communications parameters for serial ports only without changing the active modem.

Syntax

```
SETCOMM commsettings
```

Remarks

The *commsettings* parameter should be a string in the following format:

```
"baudrate,DPS"
```

baudrate is the baud rate.

D is the number of data bits.

P is the type of parity: N for None, E for Even, O for Odd, M for Mark, or S for Space.

S is the number of stop bits.

Example

This example sets the communications port to 19200 baud, 8 data bits, no parity bit, and one stop bit.

```
setcomm "19200,8N1"
```

SETDTR Statement

Sets the DTR (Data Terminal Ready) signal high (ON) or low (OFF).

Syntax

```
SETDTR on | off
```

Remarks

Most modems will disconnect when the DTR signal is lowered. This may be more reliable than using the “hangup” command.

See Also

HANGUP

Example

This example sends a command to log off an on-line service, waits five seconds, then hangs up the modem.

```
send "bye"

delay 5

setdtr off
```

SGN Function

Returns an integer indicating the sign of a number.

Syntax

```
SGN(expr)
```

Remarks

If expr is greater than zero, the return value is 1. If expr is less than zero, the return value is –1. If expr is equal to zero, the return value is zero.

See Also

ABS

Example

This example shows what happens to a negative number, zero, and a positive number with the SGN function.

```
print sgn(-5), sgn(0), sgn(3)
```

SHELL Statement

Runs a DOS command or other executable program or batch file from a script.

Syntax

```
SHELL [command]
```

Remarks

command is a string expression containing an executable DOS or Windows 95 command.

See Also

END, SYSTEM

Example

This example runs Checkdisk.

```
shell "CHKDSK.EXE "
```

SHORT Type

Used to declare a variable that can handle short numbers.

Remarks

Variables of integer type can hold values that range from -32768 to 32767.

See Also

BYTE, DIM, LONG, REAL, SHORT, TYPE

Example

This example declares a variable of type SHORT and assigns a value to it.

```
dim i as short

i = 5000

print i
```

SIN Function

Returns the sine of an angle.

Syntax

```
SIN(angle)
```

Remarks

angle is the measurement of an angle expressed in radians. You can compute radians to degrees by multiplying by 180/pi (pi is approximately 3.14159).

See Also

ATN, COS, TAN

Example

This example prints the sine of 1 radian.

```
print sin(1)
```

SIZE Statement

Used to resize the *QmodemPro for Windows 95* application window on the Windows 95 desktop.

Syntax

```
SIZE width, height
```

Remarks

width is the new width of the window on the desktop. Width values range from 0 to the width of your screen.

Height is the new height of the window on the desktop. Height values range from 0 to the height of your screen.

If you are using standard VGA with a 640x480 screen then the width values can range from 0 through 640 and height values can range from 0 through 480.

See Also

ACTIVATE, MAXIMIZE, MINIMIZE, MOVE

Example

This example changes the size of the *QmodemPro for Windows 95* application window to be 400 pixels wide and 300 pixels high.

```
SIZE 400, 300
```

SOUND Statement

Generates a tone of a specific frequency and duration.

Syntax

```
SOUND frequency, duration
```

Remarks

This statement generates a tone of a specific frequency and duration. The frequency is specified in hertz (cycles per second) and the duration is specified in seconds.

See Also

BEEP, PLAY

Example

This example sounds a 1000 Hz tone for 2/10 of a second.

```
SOUND 1000, 0.2
```

SPACE Function

Returns a string consisting of a specified number of spaces.

Syntax

```
SPACE(number)
```

Remarks

number is the number of spaces to return.

This function is identical to the SPC function.

See Also

SPC, TAB

Example

This example prints two words separated by 20 spaces.

```
print "here"; space(20); "there"
```

SPC Function

Returns a string consisting of a specified number of spaces.

Syntax

```
SPC(number)
```

Remarks

number is the number of spaces to return.

This function is identical to the SPACE function.

See Also

SPACE, TAB

Example

This example prints two phrases separated by 30 spaces.

```
print "neither here"; spc(30); "nor there"
```

SPLITSCREEN Statement

Turns split screen mode on or off.

Syntax

```
SPLITSCREEN onoff
```

Remarks

This command turns the split screen mode on or off. The operation of the script is not affected while in split screen mode (ie. the script continues to execute).

See Also

SCROLLBACK

Example

This example shows how to turn on and off the split screen mode from a script.

```
splitscreen on

splitscreen off
```

SQR Function

Returns the square root of a numeric expression.

Syntax

```
SQR(expression)
```

Remarks

expression must evaluate to a number greater than or equal to zero.

If the expression is less than zero, SQR will generate the ERR_MATH error. This error can be caught with the CATCH statement.

See Also

EXP, LOG

Example

This example prints the square root of 9.

```
print sqr(9)
```

STAMP Statement

Used to write a string of text to the log file, if it is open.

Syntax

```
STAMP string
```

Remarks

string is a string that is written to the log file. If the log file is not currently open, this statement has no effect.

See Also

CAPTURE, LOGFILE

Example

This example opens the log file, writes a stamp string to it, and closes the log file.

```
logfile on

stamp "This is a stamped string."

logfile off
```

STATIC Statement

Used to declare variables in a subroutine or function that retain their values even after the subroutine or function has returned to its caller. The variables can be used when the subroutine or function is called again without being reinitialized to zero or a blank string.

Syntax

```
STATIC var[([lowerbound TO] upperbound)] [AS type][, var[([lowerbound TO] upperbound)]
[AS type]...]
```

Remarks

The syntax of the STATIC statement is identical to that of the DIM statement. For a complete discussion of variable declarations please see the DIM statement.

See Also

DIM

Example

This example shows how a STATIC variable might be used to count the number of times a function has been called.

```
declare sub test
call test
call test

sub test
  static i as integer
  i = i + 1
  print "test has been called "; i; " times."
end sub
```

STOP Statement

Terminates execution of a script.

Syntax

```
STOP
```

Remarks

The STOP statement is similar to the END statement except that it puts a warning message on the screen after the script has terminated.

See Also

END

Example

This example uses STOP to terminate a loop.

```
dim i as integer
i = 0
do
  i = i + 1
  if i > 99 then stop
loop
```

STR Function

Converts a number to a string representation of the number.

Syntax

```
STR(number)
```

Remarks

This function returns the string representation of *number*.

This function is the opposite of the VAL function, which converts a number in string format to an integer.

See Also

VAL

Example

This example uses the STR function to convert a number to a string and count the number of digits in the number.

```
dim a as string

a = str(47)

print a; "is "; len(a); " digits long."
```

STRING Function

Repeats a character a specified number of times, and returns the results as a string.

Syntax

```
STRING(number, code)
```

or

```
STRING(number, string)
```

Remarks

number is the number of characters to output, in the range 0 to 32767.

code is the ASCII code for the character you want, in the range 0 to 255.

In the second syntax, *string* is a string and the STRING function returns the first character of *string* repeated *num* times.

See Also

CHR, SPACE

Example

This example prints a string of 50 dashes on the screen.

```
print string(50, "-")
```

STRING Type

Used to declare a variable that can handle variable length character data, or to define a string that can hold a specific number of characters.

Remarks

Variables of string type can hold values that are up to 32767 characters in length.

To declare a fixed length string that holds only a specific number of characters, use the syntax

```
STRING*n
```

where n is the number of characters you want the string to hold. Fixed length strings are useful because variable length strings are not allowed within user defined type declarations.

See Also

DIM, TYPE

Example

This example declares a string, assigns a value to it, and prints the value.

```
dim a as string

a = "Hello, world!"

print a
```

STRIPHIBIT Function

Used to get or set the current state of the “8th Bit Strip” toggle.

Syntax

```
STRIPHIBIT
```

```
or
```

```
STRIPHIBIT(onoff)
```

Remarks

There are two forms to the STRIPHIBIT function. The first form takes no arguments and simply returns the current setting. The second form sets the state of the toggle and returns the previous state.

In both the parameter and the return value, a nonzero value means the high bit is stripped from incoming characters. A zero value means the high bit is not stripped.

See Also

DUPLEX

Example

This example shows how the STRIPHIBIT function can be used in a logon script to turn on the high bit strip.

```
dim oldstrip as integer

oldstrip = striphibit(on)

waitfor "UserID"

send "123456"

striphibit oldstrip
```

SUB Statement

Allows you to define your own subprograms that do not directly return a value to the caller.

Syntax

```
SUB name[(arg AS type[, arg AS type]...)]
  ...
END SUB
```

Remarks

name is the name you assign to the subroutine.

arg is the name of a formal argument to the subroutine. Each argument must have an associated type declaration using the AS keyword.

You may define local variables within your user-defined subroutine. You can use STATIC declarations to preserve the value of individual variables across subroutine calls.

You can execute a sub-program using the CALL statement, along with any arguments you wish to pass to the sub-program.

See Also

BYVAL, CALL, DECLARE, EXIT, FUNCTION, STATIC

Example

This example defines a simple subroutine and shows two ways of calling it.

```
sub test(i as integer)

  print "in subroutine test: "; i

end sub


sub 1
call sub(5)
```

SYSTEM Statement

Closes all open files and terminates a script.

Syntax

```
SYSTEM
```

Remarks

This statement immediately ends the script and closes the *QmodemPro for Windows 95* application.

See Also

END, SHELL, STOP

Example

This example terminates *QmodemPro for Windows 95* after waiting for a key to be pressed.

```
print "press a key to exit...";
while inkey = ""
wend
system
```

TAB Function

Returns the appropriate number of spaces to move the cursor to the specified column on the screen.

Syntax

```
TAB(column)
```

Remarks

This function returns the appropriate number of spaces to move the cursor to a particular column on the screen. Note that this function only works on the actual terminal screen.

See Also

SPC

Example

This example shows how the TAB function might be used to print columnar data on the screen.

```
print "test"; tab(40); "column 40"
```

TAN Function

Returns the tangent of an angle.

Syntax

```
TAN(angle)
```

Remarks

angle is the measurement of an angle expressed in radians. You can convert radians to degrees by multiplying by 180/pi (pi is approximately 3.14159).

See Also

ATN, COS, SIN

Example

This example prints the tangent of 1 radian.

```
print tan(1)
```

K + TIME Function

Returns the current date from the computer's internal clock.

Syntax

```
TIME
```

Remarks

The TIME function returns the current time in the form specified in Windows 95 Control Panel, Regional Settings section.

See Also

DATE

Example

This example prints the current time according to Windows 95.

```
print "The time is now "; time
```

TIMEOUT Function

Used to either set or retrieve the current script timeout setting.

Syntax

```
TIMEOUT

or

TIMEOUT(seconds)
```

Remarks

When TIMEOUT is called without an argument, it returns the current script timeout value.
When TIMEOUT is called with a *seconds* argument, it sets the current timeout value to the specified number of seconds. In addition, it returns the previous timeout value.
If the timeout is set to zero, the timeout function is disabled.
The script timeout value applies to INPUT, RECEIVE, and WAITFOR commands.

See Also

INPUT, RECEIVE, WAITFOR, WHEN

Example

This example shows how you might modify the timeout value during a logon script.

```
timeout 20

waitfor "what is your first name"

send "joe"

...

catch err_timeout

hangup

end
```

TIMER Function

Returns the number of seconds since midnight.

Syntax

```
TIMER
```

Remarks

TIMER returns the number of seconds since midnight as a real type value.

See Also

TIMEOUT, WAITFOR, WHEN TIME

Example

This example prints the number of seconds since midnight, according to Windows 95.

```
print timer
```

TRAPSCREEN Statement

Used to capture the current terminal screen to a file.

Syntax

```
TRAPSCREEN filename
```

Remarks

This function takes a snapshot of the current terminal screen and appends it to the named file.   A date and time stamp precedes the data on the screen.

See Also

CAPTURE, SCROLLBACK

Example

This example shows how the TRAPSCREEN command might be used to capture some data from the remote host.

```
send

waitfor "OK"

trapscreen "snapshot.txt"
```

TYPE ... END TYPE Statement

Allows you to create a data structure (data type or user-defined type) consisting of one or more fields.

Syntax

```
TYPE typename
  field AS typename
  ...
END TYPE
```

Remarks

typename is the name of the new type.

fieldname is an element of the user-defined data type.

typename is the type of the field. It can be of type BYTE, INTEGER, LONG, REAL, or STRING*n (variable length strings are not permitted in user-defined types), or another user-defined type.

Variables of your new user-defined type are declared just like regular variables using the DIM statement. To access a field of your user defined variable, use a period after the variable name followed by the field name. See the example for an illustration of this.

See Also

DIALOG, DIM

Example

This example declares a user-defined type called “daterec”, declares a variable d of the type, assigns values to each of its fields, and prints the values of each of its fields.

```
type daterec
  day as integer
  month as integer
  year as integer
end type
dim d as daterec
d.day = 11
d.month = 10
d.year = 1993
print d.day, d.month, d.year
```

UCASE Function

Returns a copy of a string with all lower case characters converted to upper case.

Syntax

```
UCASE(string)
```

Remarks

string can be any string expression.

This operation is useful for making case insensitive comparisons of text. The LCASE function operates similarly, but converts the specified text to lower case.

See Also

LCASE

Example

This example shows how the UCASE function converts all lower case characters in a string to upper case.

```
dim a as string

a = "QmodemPro for Windows 95"

print ucase(a)
```

UPDATEPHONEENTRY Function

Updates a phonebook entry.

Syntax

```
UPDATEPHONEENTRY(entry as PhoneEntry)
```

Remarks

This function updates the phonebook entry specified by the entry parameter. The function returns True if the entry was updated successfully

See Also

ADDPHONEENTRY, REMOVEPHONEENTRY

Example

This example changes all 805 area codes to 800 in the currently open phonebook.

```
dim count as integer
dim entry as phoneentry
count = 0
do while count < getphoneentrycount
  getphoneentry (count,entry)
  print "Changing Entry #";count; : ";entry.name;" : ";entry.areacode
  if entry.areacode = "805" then
    entry.areacode = "800"
    if updatephoneentry (entry) then
      print "Update Successful"
    else
      print "Update Failed"
    end if
  else
    print "Didn't find 805 in ";entry.name
  end if
  count = count + 1
loop
```

UPLOAD Function

Used to send files to a remote computer.

Syntax

```
UPLOAD(filename, protocol)
```

Remarks

This function initiates a file transfer to send files to a remote computer. At the time this command is executed, the remote computer must already have started the file transfer. If you are connecting to a bulletin board system (BBS) then your script should already have sent the command to the BBS that will start the file transfer.

protocol is one of the following predefined constants:

ASCII, XMODEM, XMODEMCRC, XMODEM1K, XMODEM1KG, YMODEM, YMODEMG, ZMODEM, KERMIT

The first five protocols require a single file name in the *filename* parameter. This file name indicates the file to send.

The last four protocols can accept more than just one file in the *filename* parameter — separate multiple filenames with spaces.

The UPLOAD function returns zero if the file transfer was successful. If the transfer was unsuccessful, UPLOAD returns the error code describing the error. For a list of error codes see Chapter 4.

This function is identical to the SENDFILE function.

See Also

DOWNLOAD, RECEIVEFILE, SENDFILE

Example

This example starts a file transfer of “c:\qmwin\test.dat” to a remote computer. It assumes that the remote computer is prepared to receive the file.

```
if upload("c:\qmwin\test.dat", Zmodem) = 0 then
  print "file transfer ok!"
end if
```

VAL Function

Returns the numeric value of a string.

Syntax

```
VAL(string)
```

Remarks

The VAL function begins evaluating a string at the first character, and scans it until the end of the string, or until a non-numeric character is reached, whichever comes first. It returns the value of the number found in the string, if any (if no number is found, zero is returned).

The VAL function does not support REAL numbers.

See Also

STR

Example

This example converts the string containing the digits 4 and 2 to the decimal value 42.

```
dim a as string, i as integer

a = "42"

i = val(a)

print i
```

VERSION Function

Returns the *QmodemPro for Windows 95* version number as a string.

Syntax

```
VERSION
```

Remarks

In version 2.0 of *QmodemPro for Windows 95*, this function returns the string “2.0”. You can always assume that the version numbers will be increasing with respect to the string comparison operators.

Example

This example prints the current *QmodemPro for Windows 95* version number.

```
print "QmodemPro for Windows 95 "; version
```

VIEWFILE Statement

Used to invoke the internal file viewer with a specified file.

Syntax

```
VIEWFILE filename
```

Remarks

This function brings up the internal file viewer with the named file loaded ready for viewing.   Note that that script continues to run after the viewer is started.

See Also

EDITFILE, VIEWPICTURE

Example

This example shows how the VIEWFILE command might be used to view the host mode user file.

```
viewfile "host.usr"
```

VIEWPICTURE Statement

Used to invoke the internal picture viewer with a specified file.

Syntax

```
VIEWPICTURE (filename)
```

Remarks

This function brings up the internal picture viewer with the named GIF, BMP, or JPEG file loaded ready for viewing. Note that that script continues to run after the viewer is started.

See Also

EDITFILE, VIEWFILE

Example

This example shows how to use the VIEWPICTURE function to view a GIF picture in the download directory.

```
VIEWPICTURE ("C:\qmwin\DOWNLOADS\apicture.gif")
```

WAITFOR Statement

Used to wait for a particular sequence of characters to appear from the communications port.

Syntax

```
WAITFOR string [, timeout]
```

Remarks

WAITFOR suspends script execution until the specified string appears from the communications port. By default, the timeout value is whatever was last set by the TIMEOUT command. If a timeout value is specified, it is used instead of the current TIMEOUT value.

The case of letters is ignored when comparing incoming data against the specified string. However, terminal emulation escape sequences are not ignored.

If the character string does not appear within the timeout period, the ERR_TIMEOUT error is generated. This can be caught with the CATCH script statement to take corrective action.

See Also

RECEIVE, WHEN

Example

This example shows how the WAITFOR command might be used in a script to log on to an on-line service.

```
waitfor "first name"

send "john doe"

waitfor "password"

send "sesame"
```

WAITFOREVENT Function

Suspends a script execution until either an inoming call is answered or a key is pressed on the keyboard. It returns an integer,   1 if a key has been   pressed, 2 if a call is connected.

Syntax

```
WAITFOREVENT
```

Remarks

This function is intended for use by the host mode when waiting for calls.

Example

This example tells you whether a key was pressed or a call was answered.

```
if autoanswer (getmodemname (0) ) then

  print "Autoanswer is now on."

else

  print "Failed to configure modem for autoanswer."
```

WEND Statement

Used to mark the end of a WHILE ... WEND statement. See the WHILE statement for more information and examples.

WHEN CLEAR Statement

Used to remove all when conditions, a set of when conditions, or a particular when condition.

Syntax

```
WHEN CLEAR ALL
```

```
or
```

```
WHEN CLEAR name
```

Remarks

WHEN CLEAR ALL clears all the currently active WHEN conditions.

WHEN CLEAR name clears all the WHEN conditions that have the name *name*.

After a WHEN condition is cleared, it is no longer active.

See Also

WHEN DISABLE, WHEN ENABLE, WHEN

Example

This example shows how a WHEN CLEAR command might be used to remove a WHEN MATCH condition that is no longer needed.

```
when match "press enter" do send

waitfor "main menu"

when clear all
```

WHEN DISABLE Statement

Used to disable all when conditions, a set of when conditions, or a particular when condition.

Syntax

```
WHEN DISABLE ALL
```

or

```
WHEN DISABLE name
```

Remarks

WHEN DISABLE ALL disables all the currently active WHEN conditions.

WHEN DISABLE name disables all the WHEN conditions that have the name *name*.

A disabled WHEN condition can be reactivated with the WHEN ENABLE statement.

See Also

WHEN CLEAR, WHEN ENABLE, WHEN

Example

This example shows how the when condition with the name “Enter” might be disabled if it is temporarily unnecessary.

```
when name "Enter" match "press enter" do send

...

when disable "Enter"
```

WHEN ENABLE Statement

Used to enable all when conditions, a set of when conditions, or a particular when condition.

Syntax

```
WHEN ENABLE ALL
```

or

```
WHEN ENABLE name
```

Remarks

WHEN ENABLE ALL enables all the currently active WHEN conditions.

WHEN ENABLE name enables all the WHEN conditions that have the name *name*.

See Also

WHEN CLEAR, WHEN DISABLE, WHEN ENABLE, WHEN

Example

This example shows how the WHEN ENABLE command might be used to turn on a WHEN MATCH event that has been previously disabled.

```
when name "Enter" match "press enter" do send

...

when disable "Enter"

...

when enable "Enter"
```

WHEN Statement

Used set up a mechanism that will trigger a particular statement or set of statements when a certain event happens.

Syntax

```
WHEN [NAME name] {MATCH string | QUIET seconds | TIME time} DO statements

or

WHEN [NAME name] {MATCH string | QUIET seconds | TIME time} DO

  statements ...

END WHEN
```

Remarks

There are three kinds of WHEN trigger events: MATCH, QUIET, and TIME.

A MATCH event watches the communications port for a particular sequence of characters and executes the statements associated with the WHEN statement when the sequence of characters is seen. Upper and lower case characters are considered the same when comparing characters.

A QUIET event watches the activity on the communications port and executes the statements associated with the WHEN statement when there is no activity for a certain amount of time. This is often used to send an Enter whenever there is no data received from the remote computer for, say, 10 seconds.

A TIME event watches the time and executes the statements associated with the WHEN statement when a particular time of day occurs. The time of day is specified in string format, so “2:00” would be 2:00am. Hours, minutes, and seconds are accepted separated by colons. An AM or PM indicator is also accepted.

See Also

WAITFOR, WHEN CLEAR, WHEN DISABLE, WHEN ENABLE

Example

This example shows how the WHEN MATCH command might be used to skip past logon screens when waiting for a “main menu” prompt from an on-line service.

```
when match "press enter" do send

waitfor "main menu"
```

WHILE ... WEND Statement

Repeats a series of statements while a given condition is true.

Syntax

```
WHILE expression
  [statements]
WEND
```

Remarks

expression is any numeric expression or variable that evaluates to a nonzero value (true) or zero (false). A WHILE ... WEND loop will execute so long as the expression is true, then when the expression becomes false, then passes control to the statement that follows the WEND.

The WHILE ... WEND provides an alternate syntax to the DO WHILE ... LOOP statement.

Each WHILE must have a corresponding WEND. Unmatched statements will cause an error message.

See Also

DO ... LOOP, FOR ... NEXT

Example

This example shows how a WHILE loop might be used to print out the first 10 whole numbers.

```
dim i as integer
i = 0
while i < 10
  print "i is "; i
  i = i + 1
wend
```

WINVERSION Function

Returns the current Windows 95 version as a string.

Syntax

```
WINVERSION
```

Remarks

This function returns the Windows 95 version as a string in the format X.Y. X is the major version number and Y is the minor revision number.

For example, Windows 95 version 4.0 would be reported as “4.0”.

See Also

VERSION

Example

This example prints out the current Windows 95 version number.

```
print "Currently using Windows 95 "; winversion
```

XONXOFF Statement

Used to change the state of the Xon/Xoff (software flow control) setting.

Syntax

```
XONXOFF onoff
```

Remarks

This function turns on or off Xon/Xoff (software flow control) from a script.

Example

This example shows how the XONXOFF command might be used to turn on software flow control in a logon script.

```
waitfor "UserID"

send "123456"

xonxoff on
```

XOR Operator

XOR performs a bitwise logical exclusive-or operation between its operands.

Syntax

```
op1 XOR op2
```

Remarks

op1 and op2 are logical expressions or numeric values of any integral type (byte, integer, or long integer).

This is the logical truth table for the XOR operator:

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

See Also

AND, EQV, IMP, NOT, OR Operator

Example

This example shows how the XOR operator is used to do a bitwise exclusive-or operation.

```
print "5 xor 6 is 3: "; 5 xor 6
```

Opens a dialog box allowing you to select   a text filename and loads it into the Text Editor.

Copies selected text to the clipboard. The CTRL+INSERT key combination can be used for this function, also.

Pastes selected text from the clipboard. The SHIFT+INSERT key combination can be used for this function, also.

Clicking the right mouse button on the Editor screen drops down a list of the following commands:

- Cut
- Copy
- Paste
- Delete
- Open file at cursor

Copies the selected text to the clipboard and deletes the text from the document. Anything you copy to the clipboard replaces the current clipboard contents.

Deletes (removes) any marked text from the current file.

Opens a dialog box allowing you to select another file, open it, and place it within the file you are viewing. Places the new file into the current file at the cursor.

Sets the number of spaces assigned to each tab character when tab expansion is turned on. Enter the number of spaces desired in the text window. The default setting for this option is **2**.

When this is toggled on, a set number of spaces is inserted when the tab key is pressed. If this option is turned **off**, the Keep Tabs option must be turned **on**.

This option allows the tab character to be displayed when the tab key is pressed. White space is not shown. When this is toggled on, the Insert Spaces must be toggled off.

When word wrap is turned **on**, text at the right margin automatically wraps to the next line without requiring a carriage return. A checkmark appears beside the option when word wrap is turned on.

Sets the right margin for automatic word wrap. Type the desired number into the text box. For most files that will be read online, the recommended setting is 72.

When this option is turned **on**, the text in a paragraph will line up with the left margin established by the first line in the paragraph. A checkmark appears on the menu to indicate when this option is turned on.

Allows you to choose the font to be used by the Editor. Click on the text you want the Editor to use. The selected font appears in the top text box of the font selector.

Allows you to select the size of the font used by the Editor. Click on the size font that you want the editor to use, or type in the size if the font is a scaleable one.

Displays a sample of the style and size combination of the font selected for the Editor to use.

Removes the results of the last editing command you executed. This is the same command as the keystroke [CRTL] Z.

Restores the results of the last editing command removed with the undo command. This command undoes an undo. This is the same command as the keystroke [CRTL] A.

Copies selected text to the clipboard and removes it from the document. Note that anything copied to the clipboard will replace the current clipboard contents. This is the same command as the keystroke [CRTL] X.

Copies the current selection to the Windows clipboard. Note that anything copied to the clipboard will replace the current clipboard contents. This is the same command as the keystroke [CRTL] C.

Inserts the current contents of the Windows clipboard into the document. The contents will be placed at the current cursor position. This is the same command as the keystroke [CRTL] V.

Deletes (removes) selected text from the document. This is the same command as pressing the DELETE key.

Opens a Find dialog box that allows you to define specific text to search for, and search options.

Opens a dialog box that lets you search for specified text and replace it with text typed into the **Replace with** text window.

Moves the cursor to a specified line in the active document.

Opens a dialog box prompting for a directory and drive of a new file to open. The file is then loaded into the Editor.

Opens a dialog box allowing you to select a file to be loaded into the Editor and then viewed. This can also be done by using the keyboard shortcut CTRL+O.

Writes the current file to disk, overwriting the original version of the file. To save the file without overwriting the original, use the **Save As** command. The Save command can also be send by using the keystrokes [CTRL] S.

Opens a common dialog box, prompting you to enter a name, path, and drive to save your text file. This command creates a copy of an existing file without changing it.

Opens a dialog box allowing you to set options for your printer, then prints the current file. The keyboard shortcut CTRL+P can also be used to send the current file to the active printer.

Shows a preview in full page view of how the current text file picture looks as it is sent to the printer.

Opens a Windows Common Dialog Box allowing you to select a printer and determine the setup used for printing the file.

Opens the Microsoft Exchange and places the current screen into to an electronic mail message.

Toggles the toolbar view on and off. When the Toolbar View is on, a checkmark appears beside the menu item and the toolbar is visible on the screen.

Toggles the statusbar view on and off. When the Statusbar View is on, a checkmark appears beside the menu item and the statusbar is visible on the screen.

Special features are used in addition to the text editor to make editing scripts easier. The Script Editor contains all the regular functions of the editor, as well as a Compile command menu with Compile, Primary File, and Clear Primary File commands, syntax highlighting, and a Find Error command.

Executes a compile on the currently loaded script. A status box appears, allowing you to watch the progress of the compile.

Opens a dialog box allowing you to define the script to be used as the focus of the compile. This is used when more than one script is being used. When a primary file has been defined, the path and name of the script will appear beside the **Primary File** menu command.

Clears the currently defined primary file from the set up, allowing you to define a different file or no file as the primary file.

If a run time error has been found, Find Error brings up a dialog box requesting the error number, and takes you to the line in the source code containing the error.

This property sheet contains a Set Color For option, a Text Color and Background Color palette, a display option, and a preview window. You can use these options to highlight syntax in your scripts.

Choose the type of text (syntax) to be highlighted by clicking on the text type in the text box. Normal text, comments, keywords, numbers, symbols, and strings can be highlighted. The **Sample Text** immediately displays the current highlighting combination. This is changed using the **Text Color** and **Background Color** palettes.

Select the color for specified text type by clicking on that color in the color palette. **The Sample Text** window displays the change immediately. All text in your script with that syntax will appear highlighted in that color.

Select the background color (the color of the space around the text) for the text type that by clicking on that color in the color palette. **The Sample Text** window displays the change immediately. The background for all text in your script with that syntax will appear as it does in the **Sample Text** window.

The sample text appears in this window as it will appear in your script if the current selections are accepted.

Toggles the Display Syntax Highlighting option on and off. If highlighting is **on**, the text will appear in you script in the same way it does in the Sample Text window. A checkmark appears beside the command when this option is turned on.

Removes the syntax highlighting from the currently selected text type. This text is then displayed as the default black text on a white background.

Removes the syntax highlighting from all text types. All text is then displayed as the default black text on a white background.

Opens the QmodemPro for Windows 95 Editor Help Engine

Get information about the QmodemPro for Windows 95 Editor version.

Closes the Editor. Prompts to save information before closing.

Shows the last files opened. Click on the filename to open the file.

Opens a list of options that set editing commands.

Allows you to change the way tabs are handled for the Editor.

Allows you to set word wrap and margin settings.

Closes the Colors property sheet and saves changes.

Closes the Fonts property sheet and saves changes.

Closes the Editor property sheet and saves changes.

Closes the Colors property sheet without saving changes.

Closes the Fonts property sheet without saving changes.

Closes the Editor property sheet without saving changes.

Accepts changes to the Colors property sheet without closing.

Accepts changes to the Fonts property sheet without closing.

Accepts changes to the Editor property sheet without closing.

Opens the QmodemPro Editor Help engine.

Type the error number to find in the text box.

Closes the Find Error dialog box and accepts selection.

Closes the Find Error dialog box without completing the Find.

Type the line number to go to in the text box.

Finds the line number requested and places the cursor at that line number.

Closes the Go To dialog box without completing the Find.