ScriptMaker Language Commands

A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

Operators

A

B

C

D

E

M

N

O

P

Q

R

S

' and Other Comments

See Also Example

Comments in a script file are explanations of what the script does and so forth. They are set off by special characters so that the compiler ignores them.

To comment a whole line:

- Start the line with **Rem** followed by a space.

To comment a whole line or partial line:

- Start the comment with a single quotation mark (').

The compiler ignores all characters between the single quotation mark and the end of the line.

To comment more than one line:

- Start the comment with **/*** and end it with ***/** as in the C programming language. No statements can appear on the same line as the ending comment marker. The */ can be followed only by spaces and the carriage return.

Rem

Close Copy Print

Comment Example

The following example shows how to use **Rem** to make a comment:

```
REM This script performs...
```

The next examples show how to use the single quotation mark ' to make comments:

```
MsgBox "Hello, world!" 'This displays a string inside a message box

' This script performs...
```

The last example shows how **/*** and ***/** are used to make a multi-line comment:

```
MsgBox "Hello, world!" /* This displays a string
   inside a message box. The script pauses until
   the user clicks the OK button. */
```

* (multiplication) Operator

See Also Example

The * operator indicates that two numbers are to be multiplied. The result is the product of the two.

Syntax:

operand1 * *operand2*

operand1 A numeric expression for first factor.

operand2 A numeric expression for second factor.

+ (addition) Operator
- (unary minus) Operator
- (subtraction) Operator
/ (division) Operator
\ (integer division) Operator
^ (exponentiation) Operator
MOD (remainder) Operator
Numeric Operator Precedence

Close | Copy | Print

* (multiplication) Operator Example

In the following example, start is assigned the product of 4 and 5.

```
start% = 4 * 5
```

In the next example, end is assigned start multiplied by 100.

```
end% = start * 100
```

+ (addition) Operator

See Also Example

For numbers, the **+** operator indicates that two numbers are to be added. The result is the sum of the two.

Syntax:

operand1 **+** *operand2*

operand1, *operand2*	The numeric expressions to be added.

* (multiplication) Operator
- (unary minus) Operator
- (subtraction) Operator
/ (division) Operator
\ (integer division) Operator
^ (exponentiation) Operator
MOD (remainder) Operator
Numeric Operator Precedence

Close Copy Print

+ (addition) Operator Example

In the following example, z stores the sum of x and y.

```
x& = 45113
y% = 25
z& = x + y
```

+ (concatenation) Operator

See Also Example

For strings, the **+** operator is the concatenation operator. It indicates that two string expressions are to be joined into a longer string expression. The result is a single string starting with *operand1* and ending with *operand2.* You can concatenate string expressions up to a total of 32,768 characters.

Syntax:

operand1 **+** *operand2*

operand1, *operand2*	The string expressions to be joined.

Space$( )
String$( )

Close Copy Print

+ (concatenation) Operator Example

The following example illustrates concatenation. String3 becomes the concatenation of String1 and String2: "Good Morning, how are you?"

```
String1$ = "Good Morning"
String2$ = ", how are you?"
String3$ = String1 + String2
```

- (unary minus) Operator

See Also Example

Unary minus indicates that the sign of a number is to be changed. The result is the unary minus of the specified numeric expression. A negative number becomes positive, and a positive number becomes negative.

Syntax:

-*operand*

operand	A numeric expression to change the sign of.

* (multiplication) Operator
+ (addition) Operator
- (subtraction) Operator
/ (division) Operator
\ (integer division) Operator
^ (exponentiation) Operator
MOD (remainder) Operator
Numeric Operator Precedence

- (subtraction) Operator

See Also Example

The **-** operator indicates that one number is to be subtracted from another. The result is the difference between the two.

Syntax:

operand1 **-** *operand2*

Operands:

operand1	A numeric expression to be the subtrahend.
operand2	A numeric expression to be the minuend subtracted from *operand1*.

* (multiplication) Operator
+ (addition) Operator
- (unary minus) Operator
/ (division) Operator
\ (integer division) Operator
^ (exponentiation) Operator
MOD (remainder) Operator
Numeric Operator Precedence

Close | Copy | Print

- (subtraction) Operator Examples

The following example subtracts n from -32. The result is 0.

```
s% = -32 - n
```

Close Copy Print

- (unary minus) Operator Examples

The following example takes the unary minus of 32 and assign it to the variable n.

```
n% = -32
```

/ (division) Operator

See Also Example

The */* operator indicates that one number is to be divided by another number. The result is the quotient of the two.

Syntax:

operand1 / operand2

operand1 A numeric expression for dividend.

operand2 A numeric expression for divisor.

* (multiplication) Operator
+ (addition) Operator
- (unary minus) Operator
\ (integer division) Operator
^ (exponentiation) Operator
MOD (remainder) Operator
Numeric Operator Precedence

/ (division) Operator Example

In the following example, the value of z becomes 4.

```
z = 12/3
```

< (less than) Operator

See Also Example

The **<** relational operator stands for "less than." The result is true if the first expression is less than the second expression. Otherwise, the result is false. The comparison can be performed between two numbers or between two strings, but not between a number and a string.

Syntax:

expr1 **<** *expr2*

expr1, *expr2*	The numeric or string expressions to be compared.

String Comparison
<= (less than or equal to) Operator
<> (not equal to) Operator
= (equal to) Operator
> (greater than) Operator
>= (greater than or equal to) Operator

Close Copy Print

< (less than) Operator Example

The following examples show comparisons of either numeric or string expressions and their results.

```
1 < 2
'TRUE because 1 is less than 2

"alpha" <"beta"
'TRUE because a's ASCII value is less than b's

"a " <"a"
'FALSE because the longer string is greater
'than the shorter
```

<= (less than or equal to) Operator

See Also Example

The **<=** relational operator stands for "less than or equal to." The result is true if the first expression is less than or equal to the second expression. Otherwise, the result is false. The comparison can be performed between two numbers or between two strings, but not between a number and a string.

Syntax:

expr1 **<=** *expr2*

expr1, *expr2*	The numeric or string expressions to be compared.

String Comparison
< (less than) Operator
<> (not equal to) Operator
= (equal to) Operator
> (greater than) Operator
>= (greater than or equal to) Operator

Close Copy Print

<= (less than or equal to) Operator Example

The following examples show comparisons of either numeric or string expressions and their results.

```
'TRUE because 1 is less than 2
1 <= 2

'TRUE because a's ASCII value is less than b's
"alpha" <= "beta"

'FALSE because the longer string is greater than the shorter
"a " <= "a"

'FALSE because they are equal
123 <= 123
```

<> (not equal to) Operator

See Also Example

The **<>** relational operator stands for "not equal to." The result is true if the first expression is not equal to the second expression. Otherwise, the result is false. The comparison can be performed between two numbers or between two strings, but not between a number and a string.

Syntax:

expr1 **<>** *expr2*

expr1, *expr2*	The numeric or string expressions to be compared.

String Comparison
< (less than) Operator
<= (less than or equal to) Operator
= (equal to) Operator
> (greater than) Operator
>= (greater than or equal to) Operator

Close Copy Print

<> (not equal to) Operator Example

The following examples show comparisons of either numeric or string expressions and their results.

```
'TRUE because 1 is less than 2
1 <> 2

'TRUE because a's ASCII value is less than b's
"alpha" <> "beta"

'TRUE because the longer string is greater than the shorter
"a " <> "a"

'FALSE because they are equal
123 <> 123
```

= (equal to) Operator

See Also Example

The **=** relational operator stands for "equal to." The result is true if the first expression is equal to the second expression. Otherwise, the result is false. The comparison can be performed between two numbers or between two strings, but not between a number and a string.

Syntax:

expr1 **=** *expr2*

expr1, *expr2*	The numeric or string expressions to be compared.

String Comparison
< (less than) Operator
<= (less than or equal to) Operator
<> (not equal to) Operator
> (greater than) Operator
>= (greater than or equal to) Operator

= (equal to) Operator Example

The following examples show comparisons of either numeric or string expressions and their results.

```
'FALSE because 1 is less than 2
1 = 2

'FALSE because a's ASCII value is less than b's
"alpha" = "beta"

'FALSE because the longer string is greater than the shorter
"a " = "a"

'FALSE because they are equal
123 = 123
```

> (greater than) Operator

See Also Example

The **>** relational operator stands for "greater than." The result is true if the first expression is greater than the second expression. Otherwise, the result is false. The comparison can be performed between two numbers or between two strings, but not between a number and a string.

Syntax:

expr1 **>** *expr2*

expr1, *expr2*	The numeric or string expressions to be compared.

String Comparison
< (less than) Operator
<= (less than or equal to) Operator
= (equal to) Operator
<> (not equal to) Operator
>= (greater than or equal to) Operator

> (greater than) Operator Example

The following examples show comparisons of either numeric or string expressions and their results.

```
'FALSE because 1 is less than 2
1 > 2

'FALSE because a's ASCII value is less than b's
"alpha" > "beta"

'TRUE because the longer string is greater than the shorter
"a " > "a"

'TRUE because they are equal
123 > 123
```

>= (greater than or equal to) Operator

See Also Example

The **>=** relational operator stands for "greater than or equal to." The result is true if the first expression is greater than or equal to the second expression. Otherwise, the result is false. The comparison can be performed between two numbers or between two strings, but not between a number and a string.

Syntax:

expr1 **>=** *expr2*

expr1, *expr2*	The numeric or string expressions to be compared.

String Comparison
< (less than) Operator
<= (less than or equal to) Operator
= (equal to) Operator
<> (not equal to) Operator
> (greater than) Operator

>= (greater than or equal to) Operator Example

The following examples show comparisons of either numeric or string expressions and their results.

```
'FALSE because 1 is less than 2
1 >= 2

'FALSE because a's ASCII value is less than b's
"alpha" >= "beta"

'TRUE because the longer string is greater than the shorter
"a " >= "a"

'TRUE because they are equal
123 >= 123
```

\ (integer division) Operator

See Also Example

The \ operator indicates that the integer division of one number by another number is to be performed. Each operand is rounded to an integer prior to the division. The result is the integer part of the unrounded quotient.

Syntax:

operand1 \ *operand2*

operand1	A numeric expression for dividend.
operand2	A numeric expression for divisor.

* (multiplication) Operator
+ (addition) Operator
- (unary minus) Operator
- (subtraction) Operator
/ (division) Operator
\ (integer division) Operator
^ (exponentiation) Operator
MOD (remainder) Operator
Numeric Operator Precedence

\ (integer division) Operator Example

The following are examples of integer division.

```
z = 3\1.6 'Equivalent to 3\2. The result is 1.

z = 3\1.5 'Also equivalent to 3\2. The result is 1.

z = 3\1.4 'Equivalent to 3\1. The result is 3.
```

^ (exponentiation) Operator

See Also Example

The **^** operator represents exponentiation. The result is the number calculated by raising a specified base number to a specified power.

Syntax:

base **^** *exponent*

base	A numeric expression for the base number.
exponent	A numeric expression for the power or exponent.

* Operator
+ Operator
- Operator
/ Operator
\ Operator
MOD Operator
Numeric Operator Precedence

^ exponentiation) Operator Example

The following example finds 2 cubed.

```
result% = 2^3
```

The next example also finds 2 cubed. Parentheses are used because ^ has a higher precedence than +.

```
result = (1+1)^(1+2)
```

Abs( )

See Also Example

The Abs( ) function returns the absolute value of a specified number. An absolute value is always positive.

Syntax:

Abs(*exprN)*

exprN A numeric expression to determine the absolute value of.

Fix( )
Int( )
Sgn( )

Abs( ) Example

The following example uses the Abs( ) function to determine the absolute value of a numeric expression.

```
absoluteValue = Abs(x+y+z)
```

ActivateControl

Overview See Also Example

The ActivateControl statement makes the specified dialog-box or window component (also called a control) active. It is equivalent to tabbing to the specified control. For example, you may want to make a text box active, so you can send keystrokes to fill it. If a control with the specified name or ID does not exist or is not enabled, a run-time error occurs. The Recorder generates an ActivateControl statement when a control becomes active.

Syntax:

ActivateControl *name* | *ID*

name	A string expression that identifies the control. It can be the name of a button or check box, or the text in the text control that visually precedes a list box, combination box, or text box.
ID	An integer identifying the control.

SelectButton
SelectComboBoxItem
SelectListBoxItem
SetCheckBox
SetEditText
SetOption

ActivateControl Example

The following example activates the "drives:" combination box and sends it a keystroke--but only if it exists and is enabled.

```
If ComboBoxExists("drives:") Then
   If ComboBoxEnabled("drives:") Then
      ActivateControl "drives:"
      SendKeys "c"      'Go to the C drive
   End If
End If
```

If a dialog box has a custom control, you can make the control immediately preceding it active, and then send a Tab keystroke to the dialog box.

```
ActivateControl "Portrait"
SendKeys "{TAB}"
```

AND Operator

See Also Example

The AND logical operator computes the logical AND of two expressions. The result is true if both expressions are true. Otherwise, the result is false.

Syntax:

expr1 **AND** *expr2*

expr1, *expr2* The numeric, relational, or logical expressions.

If the expressions are numeric, the result is a bitwise AND of the two expressions. If either of the expressions is a floating-point number, the two expressions are converted to longs before the bitwise AND.

If...Then...Else...End If
NOT Operator
OR Operator
XOR Operator

AND Operator Example

The AND operator is usually used in a logical expression. In the following example, it is used to test that a specified number is in the range from 1 to 10, inclusive.

```
If theNumber >= 1 AND theNumber <= 10 Then
   validNumber = TRUE
End If
```

The following example tests for more than two conditions.

```
'Give free admission to children with a birthday today
If birthMonth = Month(Now( )) AND birthDay = Day(Now( )) AND age < 18 Then
   freeAdmission = TRUE
End If
```

AnswerBox( )

Overview See Also Example

The AnswerBox( ) function allows you to display a predefined dialog box that contains:

- A message that you specify.
- As many as three command buttons for which you provide labels.
- The dialog box name BASIC.

The function returns 1, 2, or 3 depending on the button clicked by the user. If you do not provide labels for any buttons, the user sees OK and Cancel, which return 1 and 2 respectively.
If the user cancels the answer box by double-clicking the close box or pressing the Esc key, the function returns 0.

Syntax:

AnswerBox(*message* [, *button* [, *button* [, *button*]])

message	A string expression for the user to respond to. The message can contain Chr$(13)+Chr$(10) (carriage return/linefeed) to separate lines.
button	A string expression containing the label of a button. If no buttons are specified, the default labels are OK and Cancel, which return 1 and 2, respectively.

The width and height of the dialog box are sized to hold the entire contents of the message that is in 8-point Helvetica font. The maximum size of the dialog box is 5/8 of the width and 3/4 of the height of the screen. If a line is too long, it wraps from one line to the next. The widest button label determines the width of the buttons.

MsgBox

AnswerBox( ) Example

In the following example, AnswerBox( ) displays a message along with three buttons.

```
...
message = "What do you want to do with this record?"
Button_Choice = AnswerBox(message, "Add", "Modify", "Delete")
```

AppActivate

Overview See Also Example

The AppActivate statement makes the specified applications main window active. The application must already be running. The statement cannot make an applications dialog boxes active and it cannot make the main window active when it is disabled, for example, because a dialog box is active.

Syntax:

AppActivate *name*

name	A string expression containing the complete name of a main window.

The WinActivate statement makes any specified window or dialog box active (unless it is disabled).

AppClose
WinActivate
WinClose

AppActivate Example

The following example makes the main window for Excel active. The statement uses the AppFind$( ) function to find the complete name of the main window. AppFind$( ) matches the partial name "excel" and returns the complete name.

```
AppActivate AppFind$("excel")
```

AppClose

Overview See Also Example

The AppClose statement closes the specified main window.

Syntax:

AppClose [*name*]

name	A string expression containing the complete name of a main window. The default is the active main window.

The WinClose statement closes any specified window or dialog box (unless it is disabled).

AppActivate
WinActivate
WinClose

AppClose Example

The following example closes the main window whose name contains the string "word". The statement uses the AppFind$( ) function to find the complete name of the main window. AppFind$( ) matches the partial name "word" and returns the complete name.

```
AppClose AppFind$("word")
```

AppFileName$( )

Overview See Also Example

The AppFileName$( ) function returns the complete DOS pathname of the executable file associated with the specified main window.

Syntax:

AppFileName$[(*name*)]

name	A string expression containing the complete name of a main window. The default is the active main window.

AppGetPosition
AppGetState( )
AppType( )

AppFileName$( ) Example

The following example returns the executable filename (C:\APPS\WINWORD\WINWORD.EXE) associated with the Microsoft Word main window.

```
DosName$ = AppFileName$(AppFind$("word"))
```

AppFind$( )

Overview See Also Example

The AppFind$( ) function returns the complete name of an open main window whose name matches the specified partial name. If no match is found, the function returns an empty string (""). If more than one matches the string you specify, the functions returns the name of the window that was most recently used or started. This is a very useful function because most of the window-manipulation statements beginning with the **App** prefix require the complete name of a main window.

Syntax:

AppFind$(*partialName*)

partialName	A string expression containing part of the name of a main window.

AppGetActive$( )
AppList
WinFind( )
WinList

AppFind$( ) Example

The following example specifies the partial name "word", which matches a window with the name "Microsoft Word - Document1".

```
fullname$ = AppFind$("word")
'Was a matching window found?
If fullname = "" Then
   'A match was not found
   MsgBox "No match!"
Else
   MsgBox fullname
End If
```

AppGetActive$( )

Overview See Also Example

The AppGetActive$( ) function returns the complete name of the active main window. If no window is active, then the function returns an empty string ("").

Syntax:

AppGetActive$( )

AppFind$( )
AppList
WinFind( )
WinList

AppGetActive$( ) Example

The following example saves the name of the active main window for later use. For example, the subroutine in which this statement appears may execute a macro and then use AppActivate to reactivate the window that was active prior to the macro.

```
appName$ = AppGetActive$( )
...
AppActivate appName
```

AppGetPosition

Overview See Also Example

The AppGetPosition statement lets you know the position and size of the specified main window. You specify the window, and the statement's parameters contain the position of the window's upper-left corner as well as its width and height after the statement is executed. If *x*, *y*, *width*, or *height* is not a variable, then the corresponding value is not retrieved.

Syntax:

AppGetPosition *x, y, width, height* [, *name*]

x, *y*	The integer variables that will contain the horizontal and vertical distances in pixels from the upper-left corner of the screen to the upper-left corner of the window. The upper-left corner of the screen is 0, 0.
width, *height*	The integer variables that will contain the horizontal and vertical size of the window in pixels.
name	A string expression containing the complete name of a main window. The default is the active main window.

AppFileName$( )
AppGetState( )
AppMove
AppSize
AppType( )

AppGetPosition Example

In the following example, the x and y locations of the upper-left corner of the active main window are retrieved, but not the width or the height.

```
Dim x%, y%
AppGetPosition x, y, 0, 0
```

AppGetState( )

Overview See Also Example

The AppGetState( ) function determines the state (maximized, minimized, or restored) of the specified main window.

Syntax:

AppGetState [(*name*)]

name A string expression containing the complete name of a main window. The default is the active main window.

The function returns one of the following constants indicating the state of the specified window:

Constant	Value
WS_MAXIMIZED	1
WS_MINIMIZED	2
WS_RESTORED	3

AppFileName$( )
AppGetPosition
AppMaximize
AppMinimize
AppMove
AppSetState
AppType( )

AppGetState( ) Example

The following example determines the state of the active application and displays the result in a message box.

```
'Get the state of the active application
state% = AppGetState
If state = WS_MAXIMIZED Then
   MsgBox "Maximized"
ElseIf state = WS_MINIMIZED Then
   MsgBox "Minimized"
Else
   MsgBox "Restored"
End If
```

AppHide

Overview See Also Example

The AppHide statement hides the specified application. If the application was active, it is no longer. In addition, the application name is removed from the list of applications that can be switched to.

Syntax:

AppHide [*name*]

name	A string expression containing the complete name of a main window. The default is the active main window.

AppMaximize
AppMinimize
AppMove
AppRestore
AppSetState
AppShow
AppSize

AppHide, AppSetState, and AppShow Example

The following example show the use of the AppHide, AppSetState, and AppShow statements. The Notepad window is maximized, hidden, taken out of hiding (with AppShow), then restored to its original size.

```
'Maximize Notepad
AppSetState WS_MAXIMIZED, "Notepad - (Untitled)"

'Hide Notepad
AppHide "Notepad - (Untitled)"

'Show Notepad
AppShow "Notepad - (Untitled)"

'Restore Notepad
AppSetState WS_RESTORED, "Notepad - (Untitled)"
```

AppList

Overview See Also Example

The AppList statement fills the specified array with the complete names of all the main windows that are currently open.

Syntax:

AppList *namesArray*

namesArray	Name of a one-dimensional string array. The array is automatically resized to hold the names of the open main windows.

After the call, use the LBound( ) and UBound( ) functions to determine the new bounds of the array, and therefore the number of names found.

AppFind$( )
AppGetActive$( )
WinFind( )
WinList

AppList Example

In the following example, the names of all the main windows are obtained, and then displayed in a message box:

```
Dim appnames$( ), namelist$

'Get the names
AppList appnames
'Carriage return/linefeed
crlf$ = Chr$(13) + Chr$(10)

numfound% = UBound(appnames) - LBound(appnames) + 1
namelist$ = Str$(numfound) + " names found:" + crlf + crlf

For i = LBound(appnames) To UBound(appnames)
   namelist = namelist + appnames(i) + crlf
Next

'Display the list
MsgBox namelist
```

AppMaximize

Overview See Also Example

The AppMaximize statement maximizes the specified main window and makes it the active main window. However, if the window is already maximized or hidden, nothing happens. The Recorder generates an AppMaximize statement when a window is maximized.

Syntax:

AppMaximize [*name*]

name	A string expression containing the complete name of a main window. The default is the name of the active main window.

The WinMaximize statement maximizes any specified window (unless it is disabled).

AppGetState( )
AppHide
AppMinimize
AppMove
AppRestore
AppSetState
AppShow
AppSize

AppMaximize, AppMinimize, AppRestore, AppSize, and AppMove Example

The following example show the use of the AppMaximize, AppMinimize, AppRestore, AppSize, and AppMove statements. The Notepad main window is maximized, minimized, restored, resized, and moved to a new location.

```
'Maximizes Notepad and makes it active
AppMaximize "Notepad"

'Minimizes Notepad
AppMinimize

'Restores Notepad
AppRestore

'Makes the Notepad window 400x300 pixels in size
AppSize 400, 300

'Moves the window to the upper-left corner of the screen
AppMove 0, 0
```

AppMinimize

Overview See Also Example

The AppMinimize statement minimizes the specified main window. However, if the window is already minimized or hidden, nothing happens. The Recorder generates an AppMinimize statement when a window is minimized.

Syntax:

AppMinimize [*name*]

name	A string expression containing the complete name of a main window. The default is the name of the active main window.

The WinMinimize statement minimizes any specified window (unless it is disabled).

AppGetState( )
AppHide
AppMaximize
AppMove
AppRestore
AppSetState
AppShow
AppSize

AppMove

Overview See Also Example

The AppMove statement moves the specified window so that its upper-left corner is in the specified location. It has no effect if the window is currently maximized. Moving the window does not change its state (active or inactive). You can specify an off-screen location. The Recorder generates an AppMove statement when a main window is moved.

Syntax:

AppMove *x, y* [, *name*]

x, *y*	The numeric expressions indicating the horizontal and vertical distances in pixels from the upper-left corner of the screen to where you want the upper-left corner of the window. The upper-left corner of the screen is 0, 0.
name	A string expression containing the complete name of a main window. The default is the name of the active main window.

AppGetPosition
AppHide
AppMaximize
AppMinimize
AppRestore
AppSetState
AppShow
AppSize

AppRestore

Overview See Also Example

The AppRestore statement restores the specified main window to the size and location it had before it was last maximized or minimized. However, if the window is either hidden or already restored, nothing happens. The Recorder generates an AppRestore statement when a main window is restored.

Syntax:

AppRestore [*name*]

name	A string expression containing the complete name of a main window. The default is the active main window.

The WinRestore statement restores any specified window (unless it is disabled).

AppGetState( )
AppHide
AppMaximize
AppMinimize
AppMove
AppSetState
AppShow
AppSize

AppSetState

Overview See Also Example

The AppSetState statement maximizes, minimizes, or restores the specified main window. Unlike AppMaximize, AppMinimize, and AppRestore, if the specified window is currently hidden, the AppSetState statement displays the window.

Syntax:

AppSetState *state* [, *name*]

state — A predefined integer constant that specifies the state for a main window:

WS_MAXIMIZED	1
WS_MINIMIZED	2
WS_RESTORED	3

name — A string expression containing the complete name of a main window. The default is the active main window.

AppGetState( )
AppHide
AppMaximize
AppMinimize
AppMove
AppRestore
AppShow
AppSize

AppShow

Overview See Also Example

The AppShow statement makes a previously hidden main window reappear.

Syntax:

AppShow [*name*]

name	A string expression containing the complete name of a main window. The default is the active main window.

AppHide
AppMaximize
AppMinimize
AppMove
AppRestore
AppSetState
AppSize

AppSize

Overview See Also Example

The AppSize statement resizes the specified main window. If the window is either maximized or minimized, the window is not resized. The Recorder generates an AppSize statement when a main window is resized.

Syntax:

AppSize *width*, *height* [, *name*]

width, *height*	The integer expressions specifying the width and height for the window in pixels.
name	A string expression containing the complete name of a main window. The default is the active main window.

AppGetPosition
AppHide
AppMaximize
AppMinimize
AppMove
AppRestore
AppSetState
AppShow

AppType( )

Overview See Also Example

The AppType( ) function returns one of the following constants indicating the platform (DOS or Windows) on which an application runs. The application is identified by the name of its main window.

TYPE_DOS	1
TYPE_WINDOWS	2

Syntax:

AppType [(*name*)]

name	A string expression containing the complete name of a main window. The default is the active main window.

AppFileName$( )
AppGetPosition
AppGetState( )

AppType( ) Example

The following example determines the platform on which the active application runs and displays the result in a message box.

```
If AppType = DOS_TYPE Then
   MsgBox "Active application is a DOS application."
Else
   MsgBox "Active application is a Windows application."
End If
```

Numeric Operator Precedence

The numeric operators in the following table are in their order of precedence, the order in which they are evaluated. For example, whatever is inside parentheses is evaluated first. Next, the exponent is evaluated followed by the negative unary operator (-). The positive unary operator (+) is not recognized. Of the remaining operators, multiplication and division precede addition and subtraction. When operators have the same precedence, they are evaluated from left to right.

OPERATOR	MEANING
( )	Parentheses; logically groups expressions.
^	Exponentiation.
-	Unary minus; changes the sign of the numeric expression.
* /	Multiplication and division.
\	Integer division.
MOD	Modulo (*exprN1* MOD *exprN2* results in the remainder of *exprN1* \ *exprN2*).
+ -	Addition and subtraction.

ArrayDims( )

See Also Example

The ArrayDims( ) function returns an integer (from 0 to 60) indicating the number of dimensions in the specified array. If the return value is 0, the array has never been dimensioned and is, therefore, empty.

Syntax:

ArrayDims(*array*)

array The name of an array.

ArraySort
Dim
LBound( )
Option Base
ReDim
UBound( )

ArrayDims( ) Example

In the following example, the ArrayDims( ) function checks an array for emptiness. This function determines emptiness in this case because the FileList statement redimensions the array.

```
'allocate empty array
Dim files$(1 to 10)

'The FileList statement searches for files with the specified extension and
redimensions the array
FileList files, "C:\*.BAT"    'fill the array

'If the array has no dimensions, no files were found
If ArrayDims(files) = 0 Then
   Exit Sub 'exit if no elements
End If
```

ArraySort

See Also Example

The ArraySort statement sorts any one-dimensional array in ascending order. If a string array is specified, the routine sorts alphabetically (using case-sensitive string comparisons). A run-time error results if an array with more than one dimension is specified.

Syntax:

ArraySort *array*

array	Variable name of a one-dimensional array.

ArrayDims
Dim
LBound( )
Option Base
ReDim
UBound( )

ArraySort Example

The following example shows the use of ArraySort to sort an array of any type.

```
'Assume DayArray is a one-dimensional array
ArraySort DayArray
```

Asc( )

See Also Example

The Asc( ) function returns an integer between 0 and 255 corresponding to the ANSI value of the first character of the specified string.

Syntax:

Asc(*exprS*)

exprS A string expression.

Chr$( )
Hex$( )
Oct$( )
Str$( )
Val( )

Asc( ) Example

To determine the ASCII value for the letter A, you could use the following:

```
asciiA% = Asc("A")
```

AskBox$( )

Overview See Also Example

The AskBox$( ) function allows you to display a predefined dialog box that contains:

- A message that you specify.
- A text box for a response from the user.
- The OK and Cancel command buttons.
- The dialog box name BASIC.

When displayed, the dialog box is sized to the width of the message, using 8-point Helvetica font, and the text box is active.
The function returns the string in the text box, or an empty string if the user cancels the dialog box.

Syntax:

AskBox$(*message* [, *contents*])

message	A string expression that the user must respond to.
contents	A string expression (with a maximum of 255 characters) used as the initial contents of the text box. The user can accept this or type in a new string. The default is an empty string.

AskPassword$( )
InputBox$( )

AskBox$( ) Example

The following example displays an AskBox dialog box with an empty text box.

```
Filename = AskBox$("File Name:")
```

The next example makes FOO.TXT the initial contents of the text box.

```
Filename = AskBox$("File Name:", "FOO.TXT")
```

AskPassword$( )

Overview See Also Example

The AskPassword$( ) function displays a predefined dialog box that contains:

- A message
- A password box (a text box that displays an asterisk for every character the user types).
- The OK and Cancel command buttons.
- The dialog box name BASIC.

The dialog box is sized to the width of the message using 8-point Helvetica font. The password box is active. This function returns the string (up to 255 characters) that the user types, or an empty string if the user cancels the dialog box.

Syntax:

AskPassword$(*message*)

message	A string expression requesting a user's password or other sensitive information.

AskBox$( )
InputBox$( )

AskPassword$( ) Example

The AskPassword$( ) functions asks for a password from the user.

```
passwd$ = AskPassword$("Enter Password:")
```

Atn( )

See Also Example

The Atn( ) function returns a number of type double containing the arctangent of the specified number.

Syntax:

Atn(*exprN*)

exprN A numeric expression.

Cos( )
Sin( )
Tan( )

Atn( ) Example

The arctangent of a number is equivalent to the inverse tangent as the following example shows.

```
'Find the tangent of PI/2
tanPI_2 = Tan(PI/2)

'Find the arctangent of the tangent of PI/2
angle = Atn(tanPI_2)    'angle should be PI/2
```

ATTR_ARCHIVE

See Also Example

ATTR_ARCHIVE is a numeric constant with a value of 32. It represents the archive attribute. Files that have changed since the last backup have their archive attributes set.
This constant is used with the FileList and SetAttr statements and the GetAttr( ) function.

ATTR_DIRECTORY
ATTR_HIDDEN
ATTR_NONE
ATTR_NORMAL
ATTR_READONLY
ATTR_SYSTEM
ATTR_VOLUME

ATTR_ARCHIVE Example

The following example retrieves the names of all the files in the current directory whose archive attribute is set as well as the names of normal files. The retrieved files are shown in a predefined dialog box that displays a list box.

```
Dim archiveNames$(1 To 100)

FileList archiveNames, "*.*", ATTR_ARCHIVE
SelectedFile = SelectBox ("Archive Files", "Select a File", archiveNames)
```

ATTR_DIRECTORY

See Also Example

ATTR_DIRECTORY is a numeric constant with a value of 16. It represents the directory attribute indicating a directory entry.
This constant is used with the FileList and SetAttr statements and the GetAttr( ) function.

ATTR_ARCHIVE
ATTR_HIDDEN
ATTR_NONE
ATTR_NORMAL
ATTR_READONLY
ATTR_SYSTEM
ATTR_VOLUME

ATTR_DIRECTORY Example

The following example retrieves all the directory names in the current directory as well as the names of normal files.

```
Dim directoryNames$(1 To 100)

FileList directoryNames, "*.*", ATTR_DIRECTORY
```

ATTR_HIDDEN

See Also Example

ATTR_HIDDEN is a numeric constant with a value of 2. It represents the hidden attribute. This constant is used with the FileList and SetAttr statements and the GetAttr( ) function.

ATTR_ARCHIVE
ATTR_DIRECTORY
ATTR_NONE
ATTR_NORMAL
ATTR_READONLY
ATTR_SYSTEM
ATTR_VOLUME

ATTR_HIDDEN Example

The following example retrieves the names of all the hidden files in the current directory as well as the names of normal files.

```
Dim hiddenNames$(1 To 100)

FileList hiddenNames, "*.*", ATTR_HIDDEN
```

ATTR_NONE

See Also Example

ATTR_NONE is a numeric constant with a value of 64. It is used with the FileList statement to mean that no attributes are set.

ATTR_ARCHIVE
ATTR_DIRECTORY
ATTR_HIDDEN
ATTR_NORMAL
ATTR_READONLY
ATTR_SYSTEM
ATTR_VOLUME

ATTR_NONE Example

The following example retrieves the names of all the files in the current directory that have no attributes set.

```
Dim noAttrs$(1 To 100)

FileList noAttrs, "*.*", ATTR_NONE
```

ATTR_NORMAL

See Also Example

ATTR_NORMAL is a numeric constant with a value of 0. It is used with the FileList and SetAttr statements and the GetAttr( ) function. With FileList, it means that any or all attributes are set; the list is returns is the same as the list that the DOS DIR command returns. With SetAttr and GetAttr, it means that no attributes have been or are to be set.

ATTR_ARCHIVE
ATTR_DIRECTORY
ATTR_HIDDEN
ATTR_NONE
ATTR_READONLY
ATTR_SYSTEM
ATTR_VOLUME

ATTR_NORMAL Example

The following example retrieves the names of all the files in the current directory, regardless of which attributes are set.

```
Dim normalNames$(1 To 100)

FileList normalNames, "*.*", ATTR_NORMAL
```

ATTR_READONLY

See Also Example

ATTR_READONLY is a numeric constant with a value of 1. It represents the read-only attribute. This constant is used with the FileList and SetAttr statements and the GetAttr( ) function.

ATTR_ARCHIVE
ATTR_DIRECTORY
ATTR_HIDDEN
ATTR_NONE
ATTR_NORMAL
ATTR_SYSTEM
ATTR_VOLUME

ATTR_READONLY Example

The following example retrieves the names of all the names of read-only files in the current directory as well as the names of normal files.

```
Dim readonlyNames$(1 To 100)

FileList readonlyNames, "*.*", ATTR_READONLY
```

ATTR_SYSTEM

See Also Example

ATTR_SYSTEM is a numeric constant with a value of 4. It represents the system file attribute. This constant is used with the FileList and SetAttr statements and the GetAttr( ) function.

ATTR_ARCHIVE
ATTR_DIRECTORY
ATTR_HIDDEN
ATTR_NONE
ATTR_NORMAL
ATTR_READONLY
ATTR_VOLUME

ATTR_SYSTEM Example

The following example retrieves the names of all the system files in the current directory as well as the names of normal files.

```
Dim systemNames$(1 To 100)

FileList systemNames, "*.*", ATTR_SYSTEM
```

ATTR_VOLUME

See Also Example

ATTR_VOLUME is a numeric constant with a value of 8. It represents the volume label.
This constant is used with the FileList and SetAttr statements and the GetAttr( ) function.

ATTR_ARCHIVE
ATTR_DIRECTORY
ATTR_HIDDEN
ATTR_NONE
ATTR_NORMAL
ATTR_READONLY
ATTR_SYSTEM

ATTR_VOLUME Example

The following example retrieves the name of the volume label as well as the names of normal files:

```
Dim volumeNames$(1 To 100)

FileList volumeNames, "*.*", ATTR_VOLUME
```

Beep

See Also Example

Occasionally you may want to alert the user with the Beep statement, which sounds a single tone through the computer's speaker.

Syntax:

Beep

MCI( )
PlayMedia
PlayMidi
PlaySound

Beep Example

The following example causes the computer to make 10 beeps in rapid succession.

```
For i = 1 To 10
   Beep
Next i
```

Begin Dialog...End Dialog

Overview See Also Example

The Begin Dialog...End Dialog construct declares and defines a dialog box template created in Dialog Editor. Declarations of controls go between the Begin Dialog and the End Dialog statements.

Syntax:

Begin Dialog *templateName*, *x, y, width, height* [, *name*]
    [*controlDeclaration*]...
End Dialog

templateName	A string expression that specifies the name of the dialog box template.
x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in dialog units. The upper-left corner of the window is 0, 0.
width, *height*	The integer expressions indicating the width and height of the dialog box in dialog units.
name	String variable or literal indicating the name of the dialog box. The default is "Untitled".
controlDeclarations	Declarations for the: OK button Cancel button other command buttons option buttons text controls text boxes list boxes combination boxes

An error is generated if the dialog box template is empty.

After the template declaration, a variable is declared as the name of an instance of the template and a Dialog( ) function (or statement) displays the instance of the template. For example:
Dim *instance_name* **As** *template_name*
selected_button = **Dialog** (*instance_name*)

Dialog
Dialog( )

Begin Dialog...End Dialog Example

The following example defines a dialog template named locateDialog. It has a text control that displays a message for the user and an OK button. The Dim statement declares myDlg as an instance of the template. Then the Dialog( ) function displays that instance.

```
Begin Dialog locateDialog 10,10,100,100, "Continue"
   Text 40,14,48,8 "Do you want to continue?"
   OkButton 64,50,45,14
End Dialog

Dim myDlg As locateDialog

i = Dialog(myDlg)
```

ButtonEnabled( )

Overview See Also Example

The ButtonEnabled( ) function returns TRUE if the specified command button is enabled in the active window or dialog box, or FALSE if the button is not enabled. A run-time error occurs if the button does not exist.
This allows you to avoid the run-time error that occurs if a statement is executed for a button that is disabled (dimmed).

Syntax:

ButtonEnabled(*name* | *ID*)

name	A string expression containing the name of the button.
ID	An integer identifying the button.

ButtonExists( )
SelectButton
CheckBoxEnabled( )
ComboBoxEnabled( )
EditEnabled( )
ListBoxEnabled( )
OptionEnabled( )

ButtonExists( ), ButtonEnabled( ), and SelectButton Example

The following example selects a command button named "Find Next" if it both exists and is enabled. This avoids run-time errors when the button does not exist or is not enabled.

```
If ButtonExists("Find Next") = TRUE Then
   If ButtonEnabled("Find Next") = TRUE Then
      SelectButton "Find Next"
   End If
End If
```

ButtonExists( )

Overview See Also Example

The ButtonExists( ) function returns TRUE if the specified button exists in the active window or dialog box, or FALSE if it does not exist.
This allows you to avoid the run-time error that occurs if a statement is applied to a button that does not exist.

Syntax:

ButtonExists(*name* | *ID*)

name	A string expression containing the name of the button.
ID	An integer that identifies the button.

ButtonEnabled( )
SelectButton
CheckBoxExists( )
ComboBoxExists( )
EditExists( )
ListBoxExists( )
OptionExists( )

Call

See Also Example

The call statement calls a subroutine. It transfers control from the routine making the call to the routine named in the call. After the called routine is executed, control returns to the calling routine, and the statement that follows the subroutine call is executed.
A function or subroutine can call any subroutine that is declared above it in the script; it can also call itself. When a function or subroutine calls itself, it is recursive.

Syntax:

[**Call**] *subName* [**(** [*parameterList*] **)**]

subName	The name of the subroutine to be called.
parameterList	List of parameters to be passed to the subroutine.

The reserved word Call optionally precedes the subroutine name in a statement that calls the subroutine. Use the reserved word Call and the parentheses around the parameters. Or, omit both the reserved word and the parentheses.

Call Example

The following examples both call a subroutine that has three parameters.

```
Call Task1 (day, hours, user)
Task1 day, hours, user
```

Each of the next two examples calls a subroutine that has no parameters.

```
Task2
Call Task2 ( )
```

The next examples both call a subroutine that has three parameters--the second of which is passed by value.

```
Call Task1 (day, (hours), user)
Task1 day, (hours), user
```

Calling a Function

See Also Example

A function can be called by any subroutine or function that is declared after it in the script; a function can also call itself. When a function calls itself, it is recursive.

You call a function by using its name in an expression in the calling routine. As the expression is executed, control is transferred to the statements in the function's declaration. One of those statements is an assignment statement that assigns a return value to the functions name. After the function is executed, control is returned to the calling routine and the calculation of the expression is completed. Therefore, using the functions name in the expression is the same as using its return value in the expression.

Syntax:

... *functionName* [**(** [*parameterList*] **)**] ...

functionName	Name of function to be called.
parameterList	List of parameters to be passed to the function.

NOTE: If a 0 or empty string ("") is returned by a function, the function may be missing the assignment statement that gives the functions name a value.

Calling a Function Example

In the following example, the functions name is SquareRoot( ).

```
x = SquareRoot(y)
```

In the function declaration, the function's name is assigned a value.

```
Function SquareRoot (someNumber as double) as double
   ...
   SquareRoot = ... 'square root of someNumber
End Function
```

During the execution of the statements in the function declaration, the parameter y becomes someNumber, its square root is calculated, and the value of that square root is assigned to the name of the function.

CancelButton

Overview See Also Example

The CancelButton statement defines a Cancel button that appears within a dialog box template. It can appear only within a Begin Dialog...End Dialog construct.

Syntax:

CancelButton *x, y, width, height*

x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in dialog units. The upper-left corner of the window is 0, 0.
width, *height*	The integer expressions indicating the width and height of the dialog box in dialog units.

Begin Dialog...End Dialog
Dialog
Dialog( )

OKButton and CancelButton Example

The following example displays an instance of a dialog template with an OK and a Cancel button. Selecting either button ends the execution of the Dialog( ) function and the dialog box is no longer displayed. If OK is selected, the Dialog( ) function returns TRUE which is equal to -1. If Cancel is selected, the function returns FALSE which is equal to 0. The result is displayed in a message box.

```
'Define the dialog box template
Begin Dialog UserDialog 15, 28, 100, 100, "OK and Cancel"
   Text 40,14,48,8 "Do you want to continue?"
   OKButton 55, 64, 41, 14
   CancelButton 55, 82, 41, 14
End Dialog

'Declare the name of the instance of the template
Dim OKCancelDialog As UserDialog

'Display the instance of the template
result = Dialog(OKCancelDialog)

'What was the result?
If result = TRUE Then
   MsgBox "OK"
Else
   MsgBox "Cancel"
End If
```

CDbl( )

See Also Example

The CDbl( ) function converts the specified number to a number of type double and returns that number. This is equivalent to assigning the number to a variable of type double. A run-time error occurs if the specified expression is not within the correct range.

Syntax:

CDbl(*exprN*)

exprN	A numeric expression within the range for numbers of type double: approximately +/- 1.7E+/-308.

CInt( )
CLng( )
CSng( )
CStr( )

CDbl( ) Example

The following two assignments have the same effect.

```
x# = CDbl(4)       'Explicit conversion
x# = 4             'Implicit conversion
```

ChDir

See Also Example

The ChDir statement changes the directory on the current drive. It is the same as the DOS CD command. If a drive is specified, the current directory of the specified drive is changed, but the current drive remains the same.

Syntax:

ChDir *newDir*

newDir A string expression containing a complete or relative pathname for the existing directory that you want to make the current directory.
The current drive is the default drive.
If the specified directory does not exist, an error occurs.

ChDrive
Kill
MkDir
Name...As
RmDir

ChDir Example

Assuming that the current drive is C, and the directory \LEVEL1\SUB1 exists on the D drive, then the following example makes that directory the current directory on the D drive.

```
ChDir "D:\LEVEL1\SUB1"
```

In the above example, the C drive remains the current drive.

To change to the directory one level above the current directory, you could use the following:

```
ChDir ".."
```

ChDrive

See Also Example

The ChDrive statement makes the specified drive the current drive. If the drive does not exist, a run-time error occurs.

Syntax:

ChDrive d*riveLetter*

driveLetter	A string expression whose first letter is the drive you want to change to.

ChDir
Kill
MkDir
Name...As
RmDir

ChDrive Example

The following example makes the C drive the current drive:

```
ChDrive "c"
```

The next example makes the E drive the current drive:

```
ChDrive "Extended"
'Only the first character matters.
```

CheckBox

Overview See Also Example

The CheckBox statement defines a check box within a dialog box template. It can appear only within a Begin Dialog...End Dialog construct.

Syntax:

CheckBox *x, y, width, height*, *name*, *.field*

x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in dialog units. The upper-left corner of the window is 0, 0.
width, *height*	The integer expressions indicating the width and height of the dialog box in dialog units.
name	A string variable or literal containing the name of the check box. The string can contain an ampersand & in front of the character to be used as an accelerator key.
.field	An integer variable used to set and/or retrieve the value of the check box. Its value is 0 for unchecked, 1 for checked, and 2 for filled.

Begin Dialog...End Dialog
Dialog
Dialog( )

CheckBox and Group Box Example

The following example displays a dialog box with two check boxes within a group box.

```
Dim checkMsg2$, chkMsg$(1)

chkMsg(0) = "unchecked!"
chkMsg(1) = "checked!"

checkMsg2 = "No, check me!"

'Declare the dialog
Begin Dialog UserDialog 15,28,100,100, "Untitled"
   GroupBox 4,4,84,51, "Check Boxes"
   CheckBox 10,15,48,14, "Check me!", .CheckBox1
   CheckBox 10,35,68,14, checkMsg2, .CheckBox2
   OKButton 55,64,41,14
End Dialog

'Declare the name for the instance of the template
Dim MyDialog As UserDialog

'Make the first check box initially checked
MyDialog.CheckBox1 = 1

'Display the instance of the template
Dialog MyDialog

'What was the result?
MsgBox "Check Box 1 was " + chkMsg(MyDialog.CheckBox1)
MsgBox "Check Box 2 was " + chkMsg(MyDialog.CheckBox2)
```

CheckBoxEnabled( )

Overview See Also Example

CheckBoxEnabled( ) returns TRUE when the check box is enabled in the active window or dialog box. It returns FALSE when the box is not enabled. A run-time error occurs if the box does not exist. This allows you to avoid the run-time error that occurs if a statement is executed for a check box that is disabled (dimmed).

Syntax:

CheckBoxEnabled(*name* | *ID*)

name	A string expression containing the name of the check box.
ID	An integer identifying the check box.

CheckBoxExists( )
SetCheckBox
GetCheckBox( )
ButtonEnabled( )
ComboBoxEnabled( )
EditEnabled( )
ListBoxEnabled( )
OptionEnabled( )

CheckBoxExists( ), CheckBoxEnabled( ), and SetCheckBox Example

The following example determines if the check box named "Wrap Title" both exists and is enabled before it checks the check box.

```
WinActivate "Control Panel|Desktop"
If CheckBoxExists("Wrap Title") = TRUE Then
   If CheckBoxEnabled("Wrap Title") = TRUE Then
      'Check the check box
      SetCheckBox "Wrap Title", 1
   End If
End If
```

CheckBoxExists( )

Overview See Also Example

CheckBoxExists( ) returns TRUE if the specified check box exists in the active window or dialog box. It returns FALSE if the specified check box does not exist. This allows you to avoid the run-time error that occurs if a statement is applied to a check box that does not exist.

Syntax:

CheckBoxExists(*name* | *ID*)

name	A string expression containing the name of the check box.
ID	An integer identifying the check box.

CheckBoxEnabled( )
SetCheckBox
GetCheckBox( )
ButtonExists( )
ComboBoxExists( )
EditExists( )
ListBoxExists( )
OptionExists( )

Chr$( )

See Also Example

The Chr$( ) function returns a string containing the character that corresponds to the specified ANSI value.

Syntax:

Chr$(*exprN*)

exprN	A numeric expression evaluating to an ANSI value between 0 and 255. A run-time error occurs if the specified value lies outside the range.

Asc( )
Hex$( )
Oct$( )
Str$( )
Val( )

Chr$( ) Example

The following example converts the ASCII value 65 to a string.

```
string65$ = Chr$(65)
```

The ASCII value for a carriage-return is 13 and the value for a linefeed is 10. The next example converts the carriage-return/linefeed characters into a string.

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

CInt( )

See Also Example

The CInt( ) function converts the specified numeric expression to an integer and returns that integer. A run-time error occurs if the specified expression is not within the correct range. The function is equivalent to assigning a numeric expression to a variable of type integer.

Syntax:

CInt(*exprN*)

exprN A numeric expression within the range for an integer (-32768 and 32767), or a run-time error occurs.

CDbl( )
CLng( )
CSng( )
CStr( )

CInt( ) Example

The following two assignments have the same effect.

```
x% = CInt(4.5)     'Explicit conversion
x% = 4.5           'Implicit conversion
```

Clipboard$

See Also Example

The Clipboard$ statement replaces anything currently inthe Windows Clipboard with the specified text string.

Syntax:

Clipboard$ *contents*

contents A string expression.

Clipboard$( )
ClipboardClear

Clipboard$ Example

The following example puts a message in the clipboard. A message box displays the contents of the clipboard for verification.

```
'Put the message in the clipboard
Clipboard$ "This is the message placed in the clipboard."
MsgBox Clipboard$( )    'Verify the placement
```

Clipboard$( )

See Also Example

The Clipboard$( ) function returns a string expression containing the contents of the Clipboard. If the Clipboard is empty or does not contain text, an empty string is returned.

Syntax:

Clipboard$( )

Clipboard$
ClipboardClear

Clipboard$( ) Example

In the following example, the contents of the clipboard are retrieved and stored in a string variable. If the clipboard was empty, a message says so. Otherwise, the contents are displayed.

```
contents$ = Clipboard$( )

'Is the clipboard empty?
If contents = "" Then
   'Empty clipboard information message
   MsgBox "The clipboard is empty.", 64
Else
   'Show the contents
   MsgBox contents
End If
```

ClipboardClear

See Also Example

To clear the contents of the clipboard, use the ClipboardClear statement.

Syntax:

ClipboardClear

Clipboard$
Clipboard$( )

ClipboardClear Example

The following example puts a message in the clipboard and uses a message box to display the contents of the clipboard for verification. Then the ClipboardClear statement clears the clipboard. A second message verifies that the clipboard has indeed been cleared.

```
'Put the message onto the clipboard
Clipboard$ "This is the message placed onto the clipboard."
MsgBox Clipboard$( )    'Verify the placement

'Clear the clipboard and verify clearance
ClipboardClear
If Clipboard$( ) = "" Then
   MsgBox "The clipboard has been cleared."
Else
   MsgBox "The clipboard has NOT been cleared."
End If
```

CLng( )

See Also Example

The CLng( ) function converts a specified number to a number of type long, which it returns. This is equivalent to assigning the number to a variable of type long. A run-time error occurs if the specified expression is not within the correct range.

Syntax:

CLng(*exprN*)

exprN A numeric expression within the range for a long (-2147483648 and 2147483647), or a run-time error occurs.

CDbl( )
CInt( )
CSng( )
CStr( )

CLng( ) Example

The following two assignments have the same effect.

```
x& = CLng(4.5)     'Explicit conversion
x& = 4.5           'Implicit conversion
```

Close

See Also Example

When you finish working with a file, it is a good idea to close the file using the Close statement. Closing a file makes sure that any updates to a file are written to disk.

Syntax:

Close [#] *fileNum* [, [#] *fileNum*]...

fileNum A numeric expression, from 0 to 255, that uniquely identifies a currently open file within your script.

The Close statement can be used to close one, more than one, or all open files. You can close:

- A single file by specifying the file number (for example, Close 1).
- Multiple files by specifying the set of file numbers to close (for example, Close 1, 2, 3, 5, 7).
- All open files by using the statement alone without any file numbers (for example, Close).

Open
Reset

Close Example

The following examples show the three possible ways to use the **Close** statement.

```
' Open five files with file numbers 1 through 5

Open "testfil1" As #1
Open "testfil2" As #2
Open "testfil3" As #3
Open "testfil4" As #4
Open "testfil5" As #5

' Now close the files

Close #3          'Closes file #3
Close #2, #4      'Closes files #2 and #4
Close             'Closes the rest of the files (#1 and #5)
```

ComboBox

Overview See Also Example

The ComboBox statement defines a combination box that appears within a dialog box template. It can appear only within a Begin Dialog...End Dialog construct.

Syntax:

ComboBox, *x, y, width, height*, *itemsArray*, *.field*

x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in dialog units. The upper-left corner of the window is 0, 0.
width, *height*	The integer expressions indicating the width and height of the dialog box in dialog units.
itemsArray	A one-dimensional string array that contains the elements to be placed into the combination box.
field	A string variable used to set and/or retrieve the string selected from the combination box. Setting this field to one of the strings from *itemsArray* gives the combination box an initial value.

Begin Dialog...End Dialog
Dialog
Dialog( )

ComboBox and ListBox Example

The following example displays a dialog box containing a combination and a list box. Both show the same array of items.

```
Dim listOfItems$(9)

'Initialize the array of items
For i = 0 To 9
   listOfItems$(i) = "Item " + Str$(i)
Next

'Declares a dialog box template
Begin Dialog ListDialog 15,24,100,84, "Lists"
   ListBox 5,5,90,48, listOfItems, .ListBox1
   ComboBox 5,65,45,100, listOfItems, .ComboBox1
   OKButton 55,64,41,14
End Dialog

'Declares an instance of the template
Dim AListDialog As ListDialog

'Displays the instance of the template
Dialog AListDialog

'Display the result
MsgBox listOfItems(AListDialog.ListBox1)
MsgBox AListDialog.ComboBox1
```

ComboBoxEnabled( )

Overview See Also Example

ComboBoxEnabled( ) returns TRUE if the specified combination box is enabled in the active window or dialog box, or FALSE if the box is not enabled. A run-time error occurs if the box does not exist. This allows you to avoid the run-time error that occurs if a statement is executed for a combination box that is disabled (dimmed).

Syntax:

ComboBoxEnabled(*name* | *ID*)

name	A string expression containing the name of the combination box. Generally, this is the text in the text control which visually precedes the combination box.
ID	An integer that identifies the combination box.

ComboBoxExists( )
SelectComboBoxItem
GetComboBoxItem( )
GetComboBoxItemCount( )
ButtonEnabled( )
CheckBoxEnabled( )
EditEnabled( )
ListBoxEnabled( )
OptionEnabled( )

ComboBoxExists( ), ComboBoxEnabled( ), and SelectComboBoxItem Example

The following example checks if the combination box named "Drives:" both exists and is enabled before it selects an item from the combination box as a default for the user.

```
If ComboBoxExists("Drives:") = TRUE Then
   If ComboBoxEnabled("Drives:") = TRUE Then
      SelectComboBoxItem "Drives:", "C"
   End If
End If
```

The following is a simple recorded example where an application is activated and then an item is selected from a combination box:

```
'Make the SuperFind application active
WinActivate "SuperFind"
'Select the [All Drives] item in
'the "Where:" combination box
SelectComboBoxItem "Where:","[All Drives]"
```

ComboBoxExists( )

Overview See Also Example

ComboBoxExists( ) returns TRUE if the specified combination box exists in the active window or dialog box, or FALSE if it does not exist. This allows you to avoid the run-time error that occurs if a statement is applied to a combination box that does not exist.

Syntax:

ComboBoxExists(*name* | *ID*)

name	A string expression containing the name of the combination box. Generally, this is the text in the text control which visually precedes the combination box.
ID	An integer that identifies the combination box.

ComboBoxEnabled( )
SelectComboBoxItem
GetComboBoxItem( )
GetComboBoxItemCount( )
ButtonExists( )
CheckBoxExists( )
EditExists( )
ListBoxExists( )
OptionExists( )

Command$( )

See Also Example

The Command$( ) function returns a string containing the parameters from the command line that started the script. Command$( ) works only with scripts that have been saved as executable files with the extension .EXE.

Syntax:

Command$[( )]

InStr( )
Item$( )
Mid$( )
Word$( )

Command$ Example

The following example shows the use of Command$( ) to obtain the command-line options (for a group and filename) required by the script.

```
'Get Group and Filename from the command line
Group_and_File$ = Command$
'Break it into the group and the filename
Lngth = Len(Group_and_File)
Brk = instr(Group_and_File," ")
If Brk = 0 Then
   'An argument must be missing
   MsgBox "Improper Arguments Supplied. Proper syntax is: AddApp Groupname
Filename", 16, "FATAL ERROR"
Else
   Group$=trim$(left$(Group_and_File, Brk-1))
   File$=trim$(right$(Group_and_File, Lngth-Brk))
...
End If
```

Conditional Constructs

See Also Example

ScriptMaker has two conditional or branching constructs: the If statement and the Select Case statement. Coming to an If...End If or Select Case...End Select construct in a script is like coming to a fork in a road. What you want to accomplish determines which fork you take, but you can take only one fork. In conditional constructs each fork is a sequence of statements. The If...End construct uses logical expressions (also called Boolean expressions) which evaluate as either true or false, to choose the sequence of statements. The sequence associated with a true expression is executed. The Select Case...End Select construct uses numeric, string, or logical expressions. The first true or matching sequence of statements is executed.

Control Constructs
If...Then...Else...End If
Select Case...End Select

Conditional Constructs Examples

The following example of an If...End If construct checks to see if the job is done or not. JobDone is a numeric variable that has been assigned either TRUE or FALSE. GoHome is a subroutine.

```
If JobDone Then
   Call GoHome
Else  ' Job is not done.
   Call DoTask
End If
```

In the following example of a **Select Case...End Select** construct, the tasks, such as DoTask1, DoTask2, are subroutines. When there are several sequences of statements to choose from, the **Select Case** statement can be easier to read and to maintain than an **If** statement.

```
Select Case TaskNumber
   Case 1   ' statements to perform if case is 1
      Call DoTask1
      ...
   Case 2   ' statements to perform if case is 2
      Call DoTask2
      ...
      ...
   Case 10  ' statements to perform if case is 10
      Call DoTask10
      ...
   Case Else
      MsgBox "No such case."
End Select
```

Const

See Also Example

Const is used to declare user-defined constants. User-defined constants are constants you create outside of functions and subroutines. This makes them global constant declarations recognized by all the user-defined functions and subroutines that follow them. Each constant is valid only in the script in which it appears.
Strings that are repeated frequently and numbers with constant values are good candidates for user-defined constants. One of the benefits of using a user-defined constant is that, if the string or number changes, you only have to update its value where it is declared.

Syntax:

Const *name* **=** *expr* [**,** *name* **=** *expr*]...

name	Name of the constant.
expr	Value to assign to the constant. It may include string or numeric literals; the predefined constants TRUE or FALSE; or previously declared user-defined constants. Functions are not allowed. You do not have to use type declarators.

Constants

Const Example

Unlike the value of a variable, the value of a constant cannot change during a script's execution. The following messages are constants because they are used repeatedly in a variety of predefined dialog boxes.

```
Const Message1 = "Are you sure?", Message2 = "Please wait..."
Sub Main
   MsgBox Message2
   ...
End Sub
```

Control Constructs

See Also

Control constructs determine what statements are executed and in what order. They control the scripts flow of execution. The following table lists ScriptMakers control constructs.

Construct	Description	Statements
Conditional constructs	Choose a sequence of statements to execute based on a criteria set within the construct.	If...End If and Select Case...End Select constructs
Loops	Execute the same sequence of statements repeatedly.	Do, While, and For...Next loops
Go-to-label constructs	Transfer control to a sequence of statements that starts with a *label*.	GoSub...Return statements that transfer control to a sequence that ends with a return statement. GoTo statements that do not require a statement at the end of the sequence. On Error statements, that can transfer control to a sequence that ends with a Resume statement.

In addition to the conditional constructs, loops, and go-to-label constructs, the Stop, End, and Sleep statements can make the execution of a script stop or pause. Furthermore, calls to functions and subroutines transfer control to the statements defined within the function or subroutine.

Cos( )

See Also Example

The Cos( ) function returns the cosine of the specified angle as a number of type double.

Syntax:

Cos(*angle*)

angle A numeric expression specifying an angle in radians.

Atn( )
Sin( )
Tan( )

Cos( ) Example

The x-coordinate of a point on a circle of radius 1 centered at the origin can be found by computing the cosine of the angle at which the point lies on the circle.

```
'Calculate the x coordinate of the point at 30 degrees
x = Cos(30*PI/180)
```

CSng( )

See Also Example

The CSng( ) function converts the specified number to a number of type single, which it returns. This is equivalent to assigning the number to a variable of type single. A run-time error occurs if the specified expression is not within the correct range.

Syntax:

CSng(*exprN*)

exprN	A numeric expression (within the range for a single: approximately +/-3.4E+/-38).

CDbl( )
CInt( )
CLng( )
CStr( )

CSng( ) Example

The following two assignments have the same effect.

```
x! = CSng(4)       'Explicit conversion
x! = 4             'Implicit conversion
```

CStr( )

See Also Example

The CStr( ) function converts a numeric expression to a string and returns that string. The first character of the string is a space if the number is positive or a minus if the number is negative.

Syntax:

CStr(*exprN*)

exprN A numeric expression.

CDbl( )
CInt( )
CLng( )
CSng( )

CStr( ) Example

The following example converts the number 4.0 to a string.

```
string40$ = CStr(4.0)
```

CurDir$( )

See Also Example

The CurDir$( ) function returns a string containing the complete directory path, including the drive letter, of the current directory for the specified drive.

Syntax:

CurDir$[(*drive*)]

drive A string expression whose first letter is used as the drive specification. The default is the current drive. Using an invalid drive letter causes an error.

DiskDrives
DiskFree( )
FileDateTime( )
FileExists( )
FileLen( )
FileParse$( )
FileType( )

CurDir$( ) Example

The following example returns the current directory on the current drive and stores the result in a string:

```
currentDirectory$ = CurDir$
```

The following example returns the current directory on the C drive and stores the result in a string:

```
currentDirectory$ = CurDir$("C")
```

Date and Time Calculations

See Also Example

When the number of days separating two dates or the number of hours separating two times needs to be calculated, the serial format proves to be very useful.

Since the serial format encodes dates as the number of days since some specified date, the number of days between two dates is just the difference between the two dates.

To calculate the number of hours between two times, you can take the difference between two serial times and multiply it by the number of hours in a day, 24.

DateSerial( )
DateValue( )
Now( )
TimeSerial( )
TimeValue( )

Date and Time Calculations Example

When a rental business rents out some item with a daily rental rate, the starting and ending dates of the rental are entered into the application, the number of days between the two dates is calculated, and then the daily rental rate is multiplied by the number of days of rental to find the total charge.

```
'startRentDate contains a serial date representing
'  the starting date of rental
'endRentDate contains a serial date representing
'  the ending date of rental
numberOfDays = endRentDate - startRentDate

'Now calculate the total charge
'  where dailyRate is the daily rental rate
totalCharge = dailyRate * numberOfDays
```

A serial time calculation might be used in a payroll application where the employees are paid an hourly rate. The following example is analogous to the one above for serial dates:

```
'timeIn contains a serial time representing
'  the starting time
'timeOut contains a serial time representing
'  the ending time
numberOfHours = (timeOut - timeIn) * 24

'Now calculate the total pay for the employee
'  where hourlyRate is the employee's hourly pay
totalPay = hourlyRate * numberOfHours
```

Date$

See Also Example

To set the system date, assign the new date to the Date$ statement.

Syntax:

Date$ = *newDate*

newDate A string expression in any of the following four formats:

- MM-DD-YYYY
- MM-DD-YY
- MM/DD/YYYY
- MM/DD/YY

MM is the month, DD is the day, and YY or YYYY is the year. When setting the date, you never need to precede single digit months or days with a zero. when using YY, the date is assumed to be in the 20th century.

Date$( )
Now( )
Time$
Time$( )
Timer( )

Date$ Example

The following two examples are equivalent.

```
Date$ = 6/02/93
```

```
Date$ = 6/2/93
```

Date$( )

See Also Example

The Date$( ) function returns the computers system date as a string in the format MM-DD-YYYY, where MM is the month, DD is the day, and YYYY is the year. If the month consists of only a single digit, a zero does not have to precede the single digit. But if the day consists of only a single digit, a zero precedes the single digit.

Syntax:

Date$[( )]

Date$
DateValue( )
Now( )
Time$
Time$( )
Timer( )

Date$( ) Example

The following example saves the current date in a string.

```
currentDate$ = Date$( )
```

DateSerial( )

See Also Example

The DateSerial( ) function returns the serial date, a number of type double representing the specified year, month, and day. It is a number of days since Dec. 30, 1899, which is the zero date.

Syntax:

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

year	A numeric expression for the year to be encoded in the serial date. If the number has only two digits, the year is assumed to be in the 20th century (for example, 45 would be 1945), otherwise the year is taken literally (for example, 2001 is the year 2001).
month	A numeric expression from 1 to 12 giving the month to be encoded in the serial date.
day	A numeric expression from 1 to 31 giving the day to be encoded in the serial date.

Date and Time Calculations
DateValue( )
Day( )
Hour( )
Minute( )
Month( )
Now( )
Second( )
TimeSerial( )
TimeValue( )
Weekday( )
Year( )

DateSerial( ) Example

To obtain the serial date for December 12, 1912:

```
serialDT# = DateSerial(12,12,12)
```

To obtain the serial date for January 1, 2010:

```
serialDT# = DateSerial(2010,1,1)
```

DateValue( )

See Also Example

The DateValue( ) function returns the serial date, a number of type double, representing the date specified in the string. It is a number of days since Dec. 30, 1899, which is the zero date.

Syntax:

DateValue(*dateStr*)

dateStr	A string expression for a date. When the function is executed from Windows, the order of the date items depends on the settings contained in the [intl] section of the WIN.INI file. Check the International dialog box from the Control Panel to review the settings. The month can be specified as a word, three-letter abbreviation (minus the period), or a number. Valid date separators are the slash (/), hyphen (-), and comma (,). Dates can contain an optional time specification, but this is not used in the formation of the returned value. If the day is missing, the first day of the month is assumed. If the year is missing, the current year is assumed.

Date and Time Calculations
Date$( )
DateSerial( )
Day( )
Hour( )
Minute( )
Month( )
Now( )
Second( )
TimeSerial( )
TimeValue( )
Weekday( )
Year( )

DateValue( ) Example

To obtain the serial date for December 12, 1912:

```
serialDT# = DateValue("12-12-12")
```

Day( )

See Also Example

The Day( ) function extracts the day of the month from a serial date. The function returns a number in the range from 1 to 31 representing the day of the serial date.

Syntax:

Day(*serialDateTime*)

serialDateTime	A serial date, a number of type double, from which the day is to be extracted.

DateSerial( )
DateValue( )
Hour( )
Minute( )
Month( )
Now( )
Second( )
TimeSerial( )
TimeValue( )
Weekday( )
Year( )

Dynamic Data Exchange (DDE) Overview

Dynamic Data Exchange (DDE) is the standard protocol by which data is exchanged between Windows applications. An application acts as a DDE client, a DDE server, or both. A DDE server offers its services to clients much as a bank offers its services to its customers. It processes client requests for information and other transactions. Its clients make requests but they do not perform any services for the server.

To conduct a *conversation* (as the exchange of data and commands is called), the server and the client must be executing at the same time. The client initiates and terminates the conversation. In ScriptMaker, you use the DDEInitiate( ) function to open a channel for the conversation and the DDETerminate or DDETerminateAll statement to end the conversation by closing a specific channel or all the open channels.

Each DDE server has:

- An application name for DDE purposes. Usually this is the name of the executable file without its extension. For example, Microsoft Excel uses Excel, Microsoft Word for Windows uses Winword, and Microsoft Windows Program Manager uses Progman. Norton Desktop for Windows passes messages for Progman on to Program Manager.
- Topic names, identifying the types of information about which it can converse. For example, Excel accepts any spreadsheet, macro, or other Excel filename as a topic name, and Winword accepts any document or template filename as a topic name. All applications that support DDE support at least one topic. Some support several.
- Item names, recognized within in each topic, that identify the types of data you can request from or send to the server application. All applications support at least one item name for each topic. For example, Excels item names are identifiers for the individual cells within a spreadsheet. You may want to know the contents of one of those cells and fill several others. You use the DDERequest( ) function to get data from the server application and the DDEPoke statement to send data to the server application.
- Command names that allow you to execute commands within the server application. For example, you can create a group in Program Manager. You use the DDEExecute statement to send a command.
- The client program waits 10 seconds (10,000 milliseconds) for a response from the server to each DDE statement. You can set a longer or shorter wait using the DDETimeOut statement.

See the users guide for the particular DDE server application you are interested in for its application name and its lists of topics, items and commands.

A ScriptMaker script can be a DDE client, but ScriptMaker cannot be a DDE server. It formats, sends, and receives DDE messages and can communicate with any executing server, but it has no applications and topics of its own.

All the application, topic, item and command names are sent as string parameters, and all the data received or sent (via DDERequest or DDEPoke) has string formats.

DDE Example

The following example is a script that has Program Manager create a new group and fill it with the executables from the current directory.

```
'*************************************************
' Install files in Program Manager group using DDE.
'*************************************************

'Adds quotation marks around command sent to progman
Function quoted(s$) As String
  quoted = Chr$(34) + s$ + Chr$(34)
End Function

Sub Main( )

'Indicates that the script continues after a run-time error
'Error handling should appear in the statement following the
'one that caused the error
On Error Resume Next

'Opens viewport to watch commands that are sent
ViewportOpen "Show DDE Example"
ViewportClear

'Returns pathname to current directory
CurrentDirectory$ = CurDir$
'Add a backslash to the end of the pathname
If Right$(CurrentDirectory, 1) <> "\" Then
   CurrentDirectory = CurrentDirectory + "\"
End If

'Creates a wildcarded DOS pathname for files ending in .EXE
DirectoryFilter$ = CurrentDirectory + "*.EXE"

'Defines dialog box named SetupDialog
Begin Dialog SetupDialog 23, 34, 223, 68, "Install Files"
   Text 5, 6, 55, 8,  "Files to install:"
   TextBox 63, 4, 150, 12,  .DirectoryFilter
   Text 5, 27, 116, 8,  "Program Manager group name:"
   TextBox 126, 25, 88, 12,  .GroupName
   OKButton 121, 46, 44, 14
   CancelButton 169, 46, 44, 14
End Dialog

'Declares Setup as a dialog box using the SetupDialog template
Dim Setup As SetupDialog

'Assigns values to text boxes in Setup
'The user can change the name of the group and the directory
Setup.GroupName = "New Group"
Setup.DirectoryFilter = DirectoryFilter

'Displays Setup for user; exits subroutine if the user selects the Cancel
```

```
button
'Dialog box allows user to change directory filter and the name of group
'BUT (to keep the example short) doesn't test for bad input from user
PushButton = Dialog(Setup)
If PushButton = 0 Then Exit Sub '0 is Cancel button

'Initiates DDE conversation with Program Manager
'Both Application and Item are "progman"
'Exits subroutine if channel not created successfully
channel = DDEInitiate("progman", "progman")
If channel = 0 Then Exit Sub

'Sends command to Program Manager to create a group with the specified name
'Run-time error occurs if the specified group already exists
cmd$ = "[CreateGroup(" + quoted(Setup.GroupName) + ")]"
DDEExecute channel, cmd$
Print cmd
If Err <> 0 Then Exit Sub

'Finds the first file that matches the filter
filename$ = Dir$(Setup.DirectoryFilter)

'Finds subsequent matches until an empty string indicates that
'there are no more matches
While filename <> ""
   cmd = "[AddItem(" + CurrentDirectory + filename$ + ")]"
   DDEExecute channel, cmd
   Print cmd
   If Err <> 0 Then Exit Sub
   filename = Dir$( )
Wend

'Ends DDE conversation by closing channel
DDETerminate channel
End Sub
```

DDEExecute

Overview See Also Example

The DDEExecute statement sends a command to a server application. If the receiving application does not execute the command, a run-time error occurs.

Syntax:

DDEExecute *channel*, *command*

channel Integer (see DDEInitiate( )) that uniquely identifies the conversation between the client and the server.

command A string expression containing the command to send. It must be in a format compatible with that application.

DDEInitiate( )
DDEPoke
DDERequest( )
DDETerminate
DDETerminateAll
DDETimeOut

DDEInitiate( )

Overview See Also Example

The DDEInitiate( ) function starts a conversation between the script and the specified application on the specified topic. It returns a unique integer representing the DDE channel that is initiated (opened). It returns zero if the link cannot be established. The DDE channel number is used in subsequent operations with that application. The link cannot be established if the specified application is not running, the topic is invalid for that application, or there is insufficient memory or system resources to establish a link.

Syntax:

DDEInitiate (*name*, *topicName*)

name	A string expression containing the name of the server application. Usually this is the name of the executable file without its extension (for example, Microsoft Excel uses Excel).
topicName	A string expression containing the topic name for the conversation (for example, Excel accepts any spreadsheet, macro, or other Excel filename as a topic name).

DDEExecute
DDEPoke
DDERequest( )
DDETerminate
DDETerminateAll
DDETimeOut

DDEPoke

Overview See Also Example

The DDEPoke statement sends data to the server application.

Syntax:

DDEPoke *channel*, *itemName*, *itemValue*

channel	Integer (see DDEInitiate( )) that uniquely identifies the conversation between the client and the server.
itemName	A string expression containing the data item to set. The format of the item name depends on the server.
itemValue	A string expression containing the new value for *itemName*.

DDEExecute
DDEInitiate( )
DDERequest( )
DDETerminate
DDETerminateAll
DDETimeOut

DDEPoke Example

The following example sends the value 1992 to row 3 column 3 of BUDGET.XLS.

```
channel = DDEInitiate("Excel", "C:\SHEETS\BUDGET.XLS")
DDEPoke channel, "R3C3", "1992"
```

To check that the data was received correctly, you can request it with DDERequest( ).

DDERequest( )

Overview See Also Example

The DDERequest( ) function lets the script retrieve data from the server application. It returns a string containing the value of the requested data item.

Syntax:

DDERequest (*channel*, *itemName*)

channel	Integer (see DDEInitiate( )) that uniquely identifies the conversation between the client and the server.
itemName	A string expression containing an application-specific name such as for a range of cells from Excel, the contents of a field or bookmark from Word for Windows, or a database field from a Windows database.

DDEExecute
DDEInitiate( )
DDEPoke
DDETerminate
DDETerminateAll
DDETimeOut

DDERequest( ) Example

You request the data in row 1, column 1 of BUDGET.XLS with the following.

```
channel = DDEInitiate("Excel", "C:\SHEETS\BUDGET.XLS")
Amt$ = DDERequest(channel, "R1C1")
```

DDETerminate

Overview See Also Example

The DDETerminate statement closes a channel and ends a specific conversation. DDETerminateAll closes all DDE channels in the script. All channels are closed by the script as it endseven without a terminate statementbut closing them yourself frees memory and channels for other uses.

Syntax:

DDETerminate *channel*

channel Integer (see DDEInitiate( )) that uniquely identifies the conversation between the client and the server that is to be terminated..

DDEExecute
DDEInitiate( )
DDEPoke
DDERequest( )
DDETerminateAll
DDETimeOut

DDETerminateAll

Overview See Also Example

DDETerminateAll closes all DDE channels in the script. All channels are closed by the script as it ends, even without a terminate statement; but closing them yourself frees memory and channels for other uses.

Syntax:

DDETerminateAll

DDEExecute
DDEInitiate( )
DDEPoke
DDERequest( )
DDETerminate
DDETimeOut

DDETerminateAll Example

In the following example, all four open DDE channels are terminated with the single DDETerminateAll statement.

```
channel1 = DDEInitiate("Excel", "C:\SHEETS\BUDGET1.XLS")
channel2 = DDEInitiate("Excel", "C:\SHEETS\BUDGET2.XLS")
channel3 = DDEInitiate("Excel", "C:\SHEETS\BUDGET3.XLS")
channel4 = DDEInitiate("Excel", "C:\SHEETS\BUDGET4.XLS")
DDETerminateAll
```

DDETimeOut

Overview See Also Example

The DDETimeOut statement sets the number of milliseconds that the script must wait for the DDE server to respond. It is valid for the DDE statements and functions that follow it.
Without this statement, the wait is 10 seconds (10,000 milliseconds).

Syntax:

DDETimeOut *numberMilliseconds*

numberMilliseconds	An integer specifying the timeout interval in milliseconds.

DDEExecute
DDEInitiate( )
DDEPoke
DDERequest( )
DDETerminate
DDETerminateAll

DDETimeOut Example

The following example sets a time limit of 3 seconds:

```
DDETimeOut 3000
```

Declare

See Also Example

An *external routine* is a subroutine, procedure, or function that exists in a file other than the one you are executing. In ScriptMaker, you can call an external routine only from a Windows Dynamic Linked Library (DLL)and only if that routine has been written to handle such calls. For example, Windows Application Program Interface (API) routines in USER.EXE, KERNEL.EXE, GDI.EXE, and so on are designed to be called from users scripts. For more information about Windows DLL routines, see the *Microsoft Windows Software Development Kit* (SDK).

To use an external routine, use a Declare statement to identify the librarys name and location. The Declare statement must precede the call to the external routine. Like Deftype and Const, the Declare statement must appear outside of any subroutine or function declaration. Declare statements are valid only during the scripts execution.

String parameters are always passed from the script to DLL routines by reference. If a DLL routin\ ifies a specified string variable, then there must be sufficient space within the string to hold the returned characters. Use the Space$( ) function to create a string of sufficient length.

Libraries containing the routines are loaded when the routine is called for the first time. Be aware that this allows a script to reference external DLLs that do not exist.

Syntax for external function:

Declare Function *functionName* [**Lib** *libName* [**Alias** *realName*]] [**(** [*parameterList*] **)**] [**As** *type*]

functionName	The name you use in your script for the external function (which may be its name in the library as well).
libName	A string literal specifying the the name of the DLL that contains the external routine. (One of the Windows API libraries in the Windows SYSTEM subdirectory: USER.EXE, KRNL386.EXE, GDI.EXE, and so forth.) The extension .EXE is assumed unless you provide another.
realName	If *name* is not the real name of the external routine as it appears within the library, this is a string literal containing the real name of the routine. Use the Alias clause when the name of an external routine contains invalid characters or conflicts with a name in your script.
parameterList	The list of parameters to be passed to the external routine. The parameter list must match the syntax of the referenced routine exactly;

	otherwise, unpredictable results may occur. By default, BASIC passes parameters by reference. When a DLL routine requires a value rather than a reference, use the ByVal reserved word to indicate this.
type	The type of value the external function returns. This is used when no type declarator is appended to the function name. If neither is used, the type is as determined by a Deftype statement or, by default, the type is an integer.

Syntax for external subroutine or procedure:

Declare Sub *subName* **Lib** *libName* [**Alias** *aliasName*] [**(** [*parameterList*] **)**]

subName	The name you use in your script for the external procedure (which may be its name in the library as well).

Calling a Function
Calling a Subroutine
External Routines

Declare Examples

All of these Declare statements allow the script to use the GetCurrentTime function in USER.EXE. The third example uses GetTime as an alias for the GetCurrentTime function because the name GetCurrentTime is already used in the ScriptMaker script that calls the function.

```
Declare Function GetCurrentTime Lib "USER"( ) As Long

Declare Function GetCurrentTime& Lib "USER"( )

Declare Function GetTime Lib "USER" Alias "GetCurrentTime" As Long
```

If a parameter need to be passed by value to the external routine, use ByVal in front of its name in the Declare statement. For example, the following C routine:

```
int MessageBeep(int);
```

is declared as follows in ScriptMaker:

```
Declare Sub MessageBeep Lib "USER" (ByVal n As Integer)
```

The following examples shows a C routine that requires a pointer to an integer. Its third parameter is declared as int far*, which is a far pointer to an integer. A pointer cannot be passed by value.

```
int SystemParametersInfo(int, int, int far*, int);
```

The next example shows how to declare this routine in ScriptMaker. Notice that ByVal is not part of the declaration of the third parameter because pointers can be passed only by reference.

```
Declare Function SystemParametersInfo Lib "user"
(ByVal action As Integer, ByVal uParam As Integer, pi_value as
integer, ByVal updateINI As Integer)
```

Strings are always passed to DLL routines by reference, too. However, the string that you pass to the external routine must be long enough to accommodate the string being returned by the called routine. Use the Space$( ) function to create a string of blank characters of an appropriate length.

```
Declare Sub GetWindowsDirectory Lib "user" (Dir$, Length%)
...
Dim windir As String
windir = Space$(128)
GetWindowsDirectory (windir,128)
```

Declaring Functions and Subroutines Example

The following example shows the declarations or definitions of the subroutines named Cube_It and Main. Main is the first subroutine to be executed. The Call statement in Main calls the Cube_It subroutine. Cube_It cubes the value of the variable *sum* that is passed to it as the parameter *x*. Since *sum* is passed by reference, changes made to its value by Cube_It are known to Main as well. In the following example, *sum* has the value 7 before the call to Cube_It and the value 343 after the call.

```
'declaration of Cube_It subroutine
Sub Cube_It (x&)
   'The variable sum becomes known to Cube_It as x
   x = x * x
End Sub

'declaration of Main subroutine
Sub Main ( )
   ...
   x = 3
   y = 4
   'sum equals 7 here
   sum = x + y
   'Execution of Cube_It occurs
   Call Cube_It (sum)
   ...'sum equals 343 here
End Sub
```

The following example shows the declarations of the Square( ) function and the Main subroutine. Each use of the functions name (Square) inside Main calls the function. A statement in Main uses Square twice in the same expression. Square is used as though it were a variable of type Long because the function is type long, and the value assigned to Square inside the Square( ) function is used to evaluate the expression inside Main. Square squares *a* (which is passed to it as a parameter the first time) and returns the value 9 to the statement. Then Square squares *b* (which is passed to it the second time) and returns the value 16 to the statement. The statement assigns the value 25 (9 + 16) to *c*.

```
'declaration of Square( ) function
Function Square (x&) As Long
   'x takes the value of the a, then b
   Square = x*x
End Function

'declaration of Main subroutine
Sub Main ( )
   ...
   a = 3
   b = 4
   'calls Square twice
   c = Square(a) + Square(b)
   ...
End Sub
```

Using Parameters in Function and Subroutine Declarations

See Also Example

The syntax for parameters used in a function or subroutine declaration is different than the syntax for parameters used in a call.

Syntax:

A *parameter list* is a series of zero or more parameters:

[*parameter* [**,** *parameter*]...]

The syntax for a parameter is:

[**ByVal**] *parameterName* [**( )**][**As** *type*]

Using the reserved word ByVal in front of a parameters name forces it to be passed to the called routine by value. Otherwise, the default mode for passing a parameter is by reference.

A parameter can be either a string, integer, long, or an array. To specify that a parameter is a simple type, use the As *Type* clause or attach a type declarator (%, & or $) to the end of the parameters name. A parameters type is never implicitly declared based on the first character in its name.

If the parameter is an array, use a pair of empty parentheses after its name.

Example of Parameters in Function and Subroutine Declarations

The following show different types of function and subroutine declarations along with their parameter declarations.

```
'The function takes an integer parameter passed by value
function IsPrime(ByVal n As Integer)
   . . .
End Function

'The subroutine takes a string array and an integer,
' both passed by reference
Sub CountElements(anArray$( ), numElements%)
   . . .
End Sub
```

Def*type*

See Also Example

You can use a Def statement to specify a simple type and the initial letters for variables of that type. The Def statements are applied when a variable's type is not specified in the Dim statement that declares it.

Syntax:

Def*type letters*

type The type of variable to be identified by its initial letter: Int for integer, Lng for long, Dbl for double, Sng for single, or Str for string.

letters A series of letters of the alphabet separated by commas. A range of letters can be specified by placing a hyphen between the first and last letters of the range. The syntax for *letters* is:

letter [**-** *letter*] [**,***letter* [**-** *letter*]]...

Def statements must appear outside of user-defined functions and subroutines, not within them. This makes them global type definitions that are valid for any subroutine or function that follows them. (It does not make the variables whose types are defined by the Def statement global variables.) Additional Def statements cannot contradict earlier ones. For example, you cannot define A-F As Integers and later define C as a string. To use the same Def statements throughout the script, make them the first statements in the script.

NOTE: If a variable does not appear in a Dim statement, nor end in a type declarator, nor start with a letter listed in a Def statement, the compiler assumes it is an integer. Because of these implicit declarations, misspellings can result in new variables you never intended. You may want to check your variable names if a script compiles successfully but does not run correctly.

More About Variables

Def*type* Example

The Def statements in the following example make any variables that are not explicitly declared into integers if their names start with I, M, or Q; into longs if their names start with A, B, C, or N; and into strings if their names start with T through Z.

```
DefInt I, M, Q
DefLng A-C, N
DefStr T-Z

Sub Main
   ...
End Sub
```

DesktopCascade

See Also Example

The DesktopCascade statement cascades all nonminimized main windows. Cascaded windows are stacked one on top of the other with enough of an offset to see the title bars of each window. If you have more windows open than fill the screen, a second stack is placed on top of the first stack.

Syntax:

DesktopCascade

DesktopSetColors
DesktopSetWallpaper
DesktopTile
IconArrange

DesktopCascade Example

Use the following example to make all nonminimized main windows cascade.

```
DesktopCascade
```

DesktopSetColors

See Also Example

The DesktopSetColors statement changes the system colors to one of the predefined schemes in the CONTROL.INI file.

Syntax:

DesktopSetColors *name*

name	A string expression containing the name of the color scheme for the desktop. A valid color scheme name can be found in either the CONTROL.INI file under the [color schemes] section, or in the Control Panel under Color settings in the Color Schemes combination box.

DesktopCascade
DesktopSetWallpaper
DesktopTile
IconArrange

DesktopSetColors Example

The following example makes Arizona the color scheme for the desktop.

```
DesktopSetColors "Arizona"
```

DesktopSetWallpaper

See Also Example

The DesktopSetWallpaper statement changes the Windows wallpaper to the specified bitmap file. If tile is TRUE, the wallpaper is tiled, otherwise it is centered on the desktop. This statement writes the new wallpaper information to the WIN.INI file. To remove the wallpaper, set *filename* to an empty string: ("").

Syntax:

DesktopSetWallpaper *filename, tile*

filename	A string expression containing the complete or a relative pathname for the bitmap file to use as wallpaper. It cannot contain wildcards (* and ?). The Windows home directory is searched if a path is not specified.
tile	A numeric expression: either TRUE or FALSE.

DesktopCascade
DesktopSetColors
DesktopTile
IconArrange

DesktopSetWallpaper Example

The following example removes the wallpaper.

```
DesktopSetWallpaper "", TRUE
```

The next example centers WINLOGO.BMP, the Windows logo, as the wallpaper.

```
DesktopSetWallpaper "winlogo.bmp", FALSE
```

The last example tiles ARCHES.BMP as the wallpaper.

```
DesktopSetWallpaper "arches.bmp", TRUE
```

DesktopTile

See Also Example

The DesktopTile statement tiles all nonminimized main windows. Tiled windows are resized and repositioned to cover the screen without overlapping.

Syntax:

DesktopTile

DesktopCascade
DesktopSetColors
DesktopSetWallpaper
IconArrange

DesktopTile Example

The following example tiles all nonminimized main windows.

```
DesktopTile
```

Dialog

Overview See Also Example

The Dialog statement displays an instance of a dialog box template. The Dialog statement ends when the user closes the dialog by pressing a command button or canceling the dialog box.

A dialog box template can be used repeatedly. This statement is just one instance of that template. Before you use this statement, you must declare the instance with the name of the template as its type.

Syntax:

Dim *instanceName* **As** *templateName*

...

Dialog *instanceName*

templateName	The name of a dialog box template declared within the script. (Its declaration starts with the Begin Dialog statement and ends with the End Dialog statement).
instanceName	The name of the instance of the dialog box template displayed by the dialog statement.

Begin Dialog...End Dialog
Dialog( )

Dialog( )

Overview See Also Example

The Dialog( ) function displays an instance of a dialog box template. The function ends when the user closes the dialog by presses a command button or canceling the dialog box. The function returns an integer indicating the button that was selected:

- -1 OK button selected.
- 0 Cancel button selected.
- >0 Command button selected. The returned number represents the button selected based on its order in the dialog template. The first defined command button has a value of 1.

~~Syntax:~~PROG_LANG_SYNTAX

Dim *instanceName* **As** *templateName*

...

Dialog *instanceName*

templateName	The name of a dialog box template declared within the script. (Its declaration starts with the Begin Dialog statement and ends with the End Dialog statement).
instanceName	The name of the instance of the dialog box template displayed by the dialog statement.

Begin Dialog...End Dialog
Dialog

Dialog Box Controls Overview

See Also

You can use ScriptMaker statements and functions to control another applications dialog boxes. The statements control command buttons, check boxes, combination boxes, list boxes, text boxes, and option buttons.

Essentially, the statements and functions do the following for each dialog-box component (also called a control):

- Determine if the control exists in the active window or dialog box. For example, the ButtonExists( ) function checks for a particular command buttons existence. Each control has a similar function.
- Determine if the control is enabled in the active window or dialog box. For example, the ComboBoxEnabled( ) function checks whether a particular combination box is enabled. Each control has a similar function.
- Activates the control using the ActivateControl statement along with the name or ID for the control.
- Selects or sets the control. For example, a check box can be unchecked using the SetCheckBox statement and a button or an item from a list box can be selected using the SelectButton statement or the SelectListBoxItem statement. Each control has a similar statement.
- Retrieves the current value of a control. For example, the GetEditText$( ) function returns the contents of a text box. Each control has a similar function. For combination boxes and list boxes, you can also retrieve the number of items in the list box using the GetListBoxItemCount( ) or the GetComboBoxListCount( ) functions.

All of these statements and functions require either the name or the ID of the dialog-box control to which the statement is applied. Each control has a unique ID, but there is no way to get that ID using ScriptMaker. IDs are accepted, nevertheless, in case you have the Microsoft Windows Software Development Kit (SDK) and want to use IDs to identify the controls.

Most of you will use only the names of the controls to identify them. For command buttons, option buttons, and check boxes, the name is the actually the name of the control. For list boxes, combination boxes, and text boxes, the text that appears immediately before the control is the name of the control, although in poorly structured dialog boxes this may not hold.

When processing the specified name, case is ignored and so is the ampersand (&) which is used to indicate the accelerator key for the control. For example, "FIND what:" and "Fi&nd What:" both match "Find What:".

If the control cannot be found or is disabled, a run-time error results. It is, therefore, good practice to test if the control exists using one of the functions that ends with Exists and if the control is enabled using one of the functions that ends with Enabled before attempting to perform some action on it.

Command Button Statements and Functions

The command button statements and functions are as follows:

ButtonEnabled( )

ButtonExists( )

SelectButton

Check Box Statements and Functions

The check box statements and functions are as follows:

CheckBoxEnabled( )

CheckBoxExists( )

GetCheckBox( )

SetCheckBox

Combination Box Statements and Functions

The combination box statements and functions are as follows:

ComboBoxEnabled( )

ComboBoxExists( )

GetComboBoxItem$( )

GetComboBoxItemCount( )

SelectComboBoxItem

Text Box Statements and Functions

The text box statements and functions are as follows:

EditEnabled( )

EditExists( )

GetEditText$( )

SetEditText

List Box Statements and Functions

The list box statements and functions are as follows:

ListBoxEnabled( )

ListBoxExists( )

GetListBoxItem$( )

GetListBoxItemCount( )

SelectListBoxItem

Option Button Statements and Functions

The option button statements and functions are as follows:

OptionEnabled( )

OptionExists( )

GetOption( )

SetOption

Dim

See Also Example

The Dim statement is used to declare simple and array variables and instances of dialog box templates. All Dim statements appear inside user-defined functions and subroutines.

To declare a simple variable, you provide an identifier and a type.

Syntax:

Dim *varName* [**As** *type*] [**,** *varName* [**As** *type*]]...

varName	The name of the variable.
type	Either a type (Integer, Long, Single, Double, or String) or the name of a dialog box template. If you use a type declarator at the end of the variables name, the [**As** *type*] clause is unnecessary. You can use both so long as they indicate the same type.

To declare an array variable, you also provide the number of dimensions and the bounds for their subscripts within parentheses.

Syntax:

Dim *arrayName* **(** [*subscripts*] **)** [**As** *type*] [**,** *arrayName* **(** [*subscripts*] **)** [**As** *type*]]...

arrayName	The name of the variable.
type	Integer, Long, Single, Double, or String. If you use a type declarator at the end of the variables name, the [**As** *type*] clause is unnecessary.
subscripts	The number of dimensions and the range of subscripts available in each dimension.

subscripts is defined as:

```
[lowerBound To ] upperBound [ , [lowerBound To] upperBound ]...
```

The number of ranges provided indicates the number of dimension.

lowerBound	A numeric expression indicating the lowest subscript in a dimension.
upperBound	A numeric expression indicating the highest subscript in a dimension.

When the **Dim** statement does not specifically set the lower bound for the subscripts in any dimension of an array, that lower bound is assumed to be 0 or the value set using Option Base.

You can declare dynamic arrays (which will be redimensioned with a ReDim statement during the script's execution) without any bounds.

```
Dim Array1 ( )
```

More About Variables
ArrayDims
ArraySort
LBound( )
Option Base
ReDim
UBound( )

Dim Example

Both of the Dim statements in the following example declare a string variable.

```
Sub Main
   Dim first_name As String   'User's first name
   Dim last_name$ 'User's last name
   ...
End Sub
```

The following example declares a one-dimensional array of type long with subscripts from 0 to 10:

```
Dim MyArray(10) As Long
```

The following example declares a two-dimensional string array with subscripts from 0 to 2 in the first dimension and from 0 to 10 in the second:

```
Dim MyStrings$(2,10)
```

The following example declares a one-dimensional string array with subscripts from 5 to x (where *x* is a numeric variable that has been declared prior to this Dim statement):

```
Dim FileNames(5 To x) As String
```

The following example declares a two-dimensional integer array (unless the Def statement specifies another type for variables beginning with the letter v) with subscripts from 1 to 10 in the first dimension and 100 to 200 in the second.

```
Dim Values(1 To 10,100 To 200)
```

Dir$( )

See Also Example

The Dir$( ) function returns a string expression containing the name of a file that is visible and matches the given file specification. Names of system and hidden files are not returned. It returns an error if a valid file specification is not used the first time the function is called.
Initially, the Dir$( ) function returns the first file that matches the specification. To find additional files after the first file with the same specification, call Dir$( ) without the *fileSpec* parameter. Each time the function is called, the next file matching the given specification is returned. If no more names match the given specification, an empty string is returned.

Syntax:

Dir$([*fileSpec*])

fileSpec A string expression containing a complete or relative pathname. The string can contain wildcards (? and *).

The first time the function is called, *fileSpec* must be specified. Subsequent calls without *fileSpec* use the most recent value for *fileSpec* as the specification for finding the next matching file.

The default is the previous file specification.

When specifying files, be aware of the following:

- If no specification is made the first time the function is called or when the function is first called after an empty string has been returned (indicating that all files matching the previous specification had been found), an error occurs.
- The function returns no names if the specification is the name of a directory (for example, "\", ".", "..", "*dirName*", where *dirName* is a directory name). To find files within a particular directory, add a specification (for example, "*.*") to the end of the directory name.
- If the specification does not contain wildcards, only the name of the file that matches the specification exactly is returned.
- Using a drive alone as the specification (for example, "C:") causes an error.
- Using "\" as the last character of the specification causes an error unless it is the first character of the path (for example, "\" or "C:\"), in which case no filenames are returned.
- The specification may contain a complete (for example, "C:\PICTURE\BITMAPS*.BMP") or a relative pathname (for example, "..\LEVEL1*.DGN").

FileDirs
FileList

Dir$( ) Example

The following example processes each file in a directory, one at a time:

```
'Find first file
file$ = Dir$("*.*")
'Check if all files have already been found
While file <> ""

   'Process another one

   'Call with no specification to find next file
   file = Dir$( )
Wend
```

DiskDrives

See Also Example

The DiskDrives statement returns an array of valid drive letters. The array consists of single character strings, and each string contains the capitalized letter of a valid drive.

Syntax:

DiskDrives *list*

list Name of a one-dimensional string array to hold the list of valid drive letters.

List can be declared either as a dynamic array, such as Dim a$( ), or as an array with one dimension such as Dim a$(1 To 100). Any other type of string variable causes an error.

The **DiskDrives** statement redimensions the array to hold exactly all the valid drive letters.

Use the LBound( ) and UBound( ) functions to determine the lower and upper subscripts for the array, and thereby the number of drive letters. Even if you declare the array with a specified lower bound, that lower bound is not guaranteed to remain the lower bound if the array has been redimensioned.

CurDir$( )
DiskFree( )
FileDateTime( )
FileExists( )
FileLen( )
FileParse$( )
FileType( )

DiskDrives Example

The following example processes all the valid drive letters:

```
'Declare a dynamic string array
Dim letters$( )

'Put all the valid drive letters into letters
DiskDrives letters
numDrivesFound% = UBound(letters) - LBound(letters) + 1
For i = LBound(letters) To UBound(letters)
   ... 'Do something with each drive letter
Next i
```

DiskFree( )

See Also Example

To determine the amount of free space available on a particular drive, use the DiskFree( ) function. It returns a number of type long indicating the number of bytes available on the specified drive.

Syntax:

DiskFree([*drive*])

drive A string expression whose first character is the drive letter. The current drive is the default.

An invalid drive letter generates an error.

CurDir$( )
DiskDrives
FileDateTime( )
FileExists( )
FileLen( )
FileParse$( )
FileType( )

DiskFree( ) Example

The following example finds the amount of free space on the current drive and stores it in the long variable freeSpace:

```
freeSpace& = DiskFree( )
```

The next example finds the amount of free space on the C drive and stores it in the long variable moreFreeSpace:

```
moreFreeSpace& = DiskFree("C")
```

Do...Loop

See Also Example

The Do...Loop has three syntaxes:

Do
[*statement*]...
Loop { **While** | **Until** }*exprL*

Do { **While** | **Until** }*exprL*
[*statement*]...
Loop

Do
[*statement*]...
Loop

exprL	A relational or logical expression.
statement	An executable statement.

A **Do** statement with a **While** in it terminates when the logical expression becomes false. A **Do** statement with an **Until** in it terminates when the expression becomes true.
If the expression is at the beginning of the **Do** loop, the loop checks the expression before it executes any of the statements inside the loop. If the expression is at the end of the **Do** loop, the loop executes the statements at least once no matter what value the logical expression has.

A Do loop without a While or an Until statement is a general looping construct. It terminates with the Exit Do statement, which appears in a conditional construct. The advantage of using this type of Do loop is that the expression that controls the loop does not have to be at the beginning or end of the loop. The disadvantage is that this type of Do loop is less structured and harder to maintain.

NOTE: If you are running the script in the editor when an infinite loop occurs, choose Abort from the Script menu to stop script execution.

For...Next
Looping Constructs
While...Wend

Do...Loop Example

The following example calculates the factorial of an integer greater than zero. In the following example, the statements in the Do loop execute at least once. If you are sure that you have to execute the statements inside the loop at least once, you can use a Do loop with the expression at the end. The loop terminates when the input FactNum is greater than or equal to 0. The While loop terminates when the counter is greater than FactNum. The value of FactNum in the first loop and the value of Counter in the second change during every iteration of the loop.

```
Sub FactCal
Dim Counter As Integer  ' For loop counter.
Dim Factorial As Integer      ' Stores the result of factorial.
Dim FactNum As Integer  ' Input number for calculation.

Do
   '  get number greater than zero
   FactNum = Val(InputBox$ ("Enter a positive integer."))
   If FactNum <= 0 Then
      MsgBox "Try again"
   End If
Loop Until FactNum > 0  ' get a positive integer.
' Now FactNum is greater than or equal to zero. Initialization:
Factorial   = 1
Counter     = 1

' Calculate factorial. When Counter is greater than
' FactNum, the While loop terminates.
While Counter <= FactNum
   Factorial = Factorial * Counter
   Counter = Counter + 1
Wend
MsgBox "The factorial is:  " + Str$(Factorial) + "."
End Sub
```

DoEvents[( )]

See Also Example

Use DoEvents when you want a script that is running in exclusive mode to yield control to other applications for multitasking and return to exclusive mode when control returns to the script. The DoEvents statement is the only way other applications can multitask when a script is running in exclusive mode.

Syntax:

DoEvents[( )]

DoEvents can be used either as a statement or a function. If used as a function, **DoEvents** returns the value 0.

exclusive

DoEvents[( )] Example

The next example shows how to use **DoEvents** to yield control to other applications for multitasking while the script is running in exclusive mode. It assumes you are looping through iterations of a time-consuming computation. The example uses the **DoEvents** statement to stop between iterations and allow Windows to execute other applications.

```
Exclusive TRUE    'Enter exclusive mode
For i = 1 To 100
   ...
   DoEvents 'Give other applications a chance to be processed
Next i
Exclusive FALSE   'Leave exclusive mode
```

DoKeys

See Also Example

The DoKeys statement sends keystrokes to the active Windows application. The Recorder generates a DoKeys statement when no mouse events occur between statements that cannot go into the event queue such as WinActivate. DoKeys does not use the event queue. It operates a local queue of its own that contains no mouse events or partial keystrokes. It does not require a QueFlush (or any other) statement to empty it.
ScriptMaker optimizes macros. If the event queue has to be used for other reasons, QueKeys is used instead of DoKeys.

Syntax:

DoKeys *keyStr*

keyStr	A string expression containing the names of full keystrokes.. Keystroke Specification Format describes the format for specifying the keystrokes.

NOTE: The **DoKeys**, SendKeys, and QueKeys statements are all very similar. They specify keystrokes to be processed in the Windows environment. QueKeys places keystrokes into the event queue and the QueFlush statement causes them to be processed. SendKeys, unlike DoKeys and QueKeys, is not recorded by the Recorder. It must be added manually to the script.

DoKeys Example

The following examples sends ten copies of the letter *a* to the active Windows application.

```
DoKeys "{a 10}"
```

EditEnabled( )

Overview See Also Example

EditEnabled( ) returns TRUE if the text box is enabled in the active window or dialog box and FALSE if the text box is dimmed. This allows you to avoid the run-time error that occurs if a statement is executed for a text box that is disabled (dimmed). If the text box does not exist in the current dialog box, a run-time error occurs.

Syntax:

EditEnabled(*name* | *ID*)

name	A string expression containing the name of the text box. Generally, this is the text in the text control which visually precedes the text box.
ID	An integer that identifies the text box.

EditExists( )
SetEditText
GetEditText$( )
ButtonEnabled( )
CheckBoxEnabled( )
ComboBoxEnabled( )
ListBoxEnabled( )
OptionEnabled( )

EditExists( ), EditEnabled( ), and SetEditText Example

The following example checks if the text box named "File Name" both exists and is enabled before it puts text into the text box.

```
If EditExists("File Name:") = TRUE Then
   If EditEnabled("File Name:") = TRUE Then
      SetEditText "File Name:", "AUTOEXEC.BAT"
   End If
End If
```

In the following example, the contents of an text box are changed:

```
WinActivate "Control Panel|Desktop"
'Set the text in the "Delay:" edit control to "12"
SetEditText "Delay:", "12"
```

EditExists( )

Overview See Also Example

EditExists( ) checks for the existence of a text box with the specified name or ID in the active window or dialog box. This allows you to avoid the run-time error that occurs if a statement is applied to a text box that does not exist. The function returns TRUE if the text box exists and FALSE otherwise.

Syntax:

EditExists(*name* | *ID*)

name	A string expression containing the name of the text box. Generally, this is the text in the text control that visually precedes the text box.
ID	An integer that identifies the text box.

EditEnabled( )
SetEditText
GetEditText$( )
ButtonExists( )
CheckBoxExists( )
ComboBoxExists( )
ListBoxExists( )
OptionExists( )

End

See Also Example

A script normally terminates after it executes the last line of the Main subroutine. If you need to stop execution earlier (perhaps because of an error that has occurred), you can use the **End** statement. It closes any open files or DDE channels before stopping the scripts execution.

Syntax:

End

Control Constructs
Stop

End Example

In the following example, the user enters a password. If after three attempts, the password has not been entered correctly, the whole script terminates using the End statement.

```
i% = 0
Do
   s$ = AskPassword$("Type in the password:")
   If s$ = "password" Then
      Exit Do
   End If
   i = i + 1
   If i = 3 Then
      End
   End If
Loop
```

Environ$( )

See Also Example

The Environ$( ) function locates an environment variable by its name or its position in the environment. If the variable is found, the function returns a string in the following format:

variable = value

Otherwise, it returns an empty string.

Syntax:

Environ$(*var* | *varNum*)

var	A string expression containing the name of the environment variable.
varNum	A numeric expression containing the integer representing the variable's position in the environment. The first variable is at variable position number 1.

ReadINI$( )
ReadINISection
WriteINI

Environ$( ) Example

The following example uses the Environ$( ) function to process each of the environment variables present in the environment:

```
'Initialize the count to 1 for the first variable
count% = 1

'Get the first environment variable
envVar$ = Environ$(count)

'Check if the last variable has already been found
While envVar <> ""
   . . . 'Process the variable and its value

   'Increment the count
   count = count + 1
   'Get the next environment variable
   envVar = Environ$(count)
Wend
```

EOF( )

See Also Example

The EOF( ) function returns the value TRUE if the end of the specified file has been reached and the value FALSE if it has not. Reading beyond the end of a file causes a run-time error, so you should check for the end of the file using the EOF( ) function.

Syntax:

EOF(*fileNum*)

fileNum	A numeric expression, from 0 to 255, that uniquely identifies a currently open file within your script.

FileAttr( )
Loc( )
LOF( )
Open
Seek
Seek( )

EOF( ) Example

The file NAMEFILE contains a list of names as follows:

```
"John","Jane","Smith","Mary"
```

The following reads the names from the file into the string array NAMES and stops when the end of file has been reached.

```
Dim NAMES(1 to 100) As String

Open "namefile" For Input As #5
i% = 1

'While the end of the file has not been reached
While not EOF(5)
   'Read in a name
   Input #5, NAMES(i)

   'Increment the subscript
   i = i + 1
Wend
Close #5
```

Erl( )

See Also

The Erl( ) function is included for compatibility with other BASICs. ScriptMaker can execute scripts written in other BASICs that use this function, but the function always returns the number 0.

Syntax:

Erl [( )]

Err
Error
Error$( )
On Error
Resume

Err

See Also Example

You can use the Err statement to set the scripts error value to a specific error number. This does not simulate the occurrence of the error in the same way as the Error statement, but the next time you test for an error, the number you assigned to Err is used (unless another error occurs in the meantime).

Syntax:

Err = *errorNum*

errorNum	An integer representing an error number.

Erl( )
Err( )
Error
Error$( )
On Error
Resume

Err Example

The following example displays the messages for errors whose numbers range from 300 to 325.

```
...
For counter% = 300 To 325
   Err = counter
   MsgBox Error$
Next counter
```

Err( )

See Also Example

The Err( ) returns the error number, an integer, for the most recently trapped error. It can be used only while an On Error statement is valid. The value of Err( ) is 0 when the script starts. It is reset to 0 by the Resume statement and when a subroutine or function ends.

Syntax:

Err [( )]

Err
Erl( )
Error
Error$( )
On Error
Resume

Error

See Also Example

The Error statement simulates a ScriptMaker run-time error or a user-defined error. If no error handling routine exists, this statement generates an error message and stops the script's execution.

Syntax:

Error *errorNum*

errorNum	A predefined error or user-defined error number.

Err
Err( )
Erl( )
Error$( )
On Error
Resume

Error Example

When errors are not being trapped, the **Error** statement stops the execution of the script and causes its error message to be displayed on the screen.

```
Error 6     'External function/subroutine does not exist
```

Error$( )

See Also Example

The Error$( ) function returns the text corresponding to the specified error number or to the most recently trapped error. It returns an empty string ("") if no run-time error has occurred. Returns "Unknown or user error code" for user-defined errors.

Syntax:

Error$ [(*errorNum*)]

errorNum	Any error number (an integer). The default is the most recently trapped error.

Err
Err( )
Erl( )
Error
On Error
Resume

Exclusive

See Also Example

Normally, Windows applications can multitask. For example, you can run a ScriptMaker script, Word for Windows, and Excel simultaneously. Windows allows one application to execute for an interval of time and then switches to another. However, running a ScriptMaker script in exclusive mode stops Windows from giving other applications their opportunities to execute. This allows the ScriptMaker script to run faster. You use the Exclusive statement to turn exclusive mode on and off. For example, you may want to turn exclusive mode on before doing a series of statements that are very time-consuming or memory-intensive and turn exclusive mode off after those statements have been executed.

Syntax:

Exclusive *state*

state	A numeric expression that makes the script run in exclusive mode when TRUE, or turns off exclusive mode when FALSE.

CAUTION: Use caution when you run a script in exclusive mode. If the script happens to be in an infinite loop and the script is in exclusive mode, you cant abort the script if the computer hangs. If you run a script in exclusive mode from the ScriptMaker Editor, you cannot use the Editors ABORT command to stop the script.

You can interrupt exclusive mode and return to it using the DoEvents statement or function. DoEvents allows Windows to cycle through the intervals for the other active applications, but as soon as the ScriptMaker script gets its turn to execute, exclusive mode takes over once again.

DoEvents[( )]

Exclusive Example

The following example shows how to use exclusive mode while doing a time-consuming computation.

```
Exclusive TRUE          'Enter exclusive mode
   ... 'A time-consuming computation
Exclusive FALSE         'Leave exclusive mode
```

Exit Do

See Also Example

You can exit a Do...Loop at any point within the loop by using an Exit Do statement. The statement usually occurs in an If statement or Select Case statement. An Exit Do statement can provide an early exit from any Do loop, but if the Do loop does not use While or Until with a logical expression, it is the expected way to exit the loop.

Syntax:

Exit Do

NOTE: An Exit Do statement can exit more than one loop. For example, if a While loop is nested inside a Do loop, the Exit Do statement exits both of them, transferring control to the first statement after the Do loop.

Exit For
Looping Constructs

Exit Do Example

The following example shows a Do loop with an Exit Do statement.

```
Do
   'sequence of statements
   If Err ( ) = BigProblem
      Exit Do
   End If
   'sequence of statements
Loop While JobNotDone
```

Exit For

See Also Example

You can exit a For...Next loop at any point within the loop by using an Exit For statement. The statement usually occurs in an If statement or Select Case statement. An Exit For statement provides a way to exit the For loop before the specified number of iterations occur.

Syntax:

Exit For

NOTE: An Exit For statement can exit more than one loop. For example, if a While loop is nested inside a For loop, the Exit For statement exits both of them, transferring control to the first statement after the For loop.

Exit Do
Looping Constructs

Exit For Example

The following example shows how a **For** loop is ended from a point within the loop.

```
For counter = 1 To 25
   ...
   If some_condition Then
      Exit For
   End If
   ...
Next
```

Exit Function

See Also Example

Normally the execution of a called function ends with the End Function statement. However, you can abort the execution of a function earlier by including an Exit Function statement in the declaration. For example, if an error occurs, you may want to return to the calling routine without finishing the called functions task.

Syntax:

Exit Function

Exit Sub
Function...End Function

Exit Function Example

The following example computes a factorial. If the parameter is negative, Exit Function is used to end the function.

```
Function Factorial(n%)
   If n < 0 Then
      Exit Function
   End If

   result% = 1

   For i = 1 To n
      result = result * i
   Next

   Factorial = result
End Function
```

Exit Sub

See Also Example

Normally the execution of a called subroutine ends with an End Sub statement. However, you can abort the execution of a routine earlier by including an Exit Sub statement in the declaration. For example, if an error occurs, you may want to return to the calling routine without finishing the called routines task.

Syntax:

Exit Sub

Exit Function
Sub...End Sub

Exit Sub Example

The following subroutine computes a factorial. If the first parameter is negative, Exit Sub is used to end the subroutine.

```
Sub Factorial(n%, result%)
   If n < 0 Then
      Exit Sub
   End If

   result = 1

   For i = 1 To n
      result = result * i
   Next
End Sub
```

Exp( )

See Also Example

The Exp( ) function calculates the result of the base *e* raised to a specified power. It returns the result as a number of type double.

Syntax:

Exp(*exprN*)

exprN	A numeric expression giving the power to which to raise the base *e*. Its range is from 0 to 709.782712893.

Log( )

Exp( ) Example

The following example calculates the value of the base *e* by raising it to the first power using the Exp( ) function.

```
e# = Exp(1)
```

External Routines

See Also Example

An *external routine* is a subroutine, procedure, or function that exists in a file other than the one you are executing. In ScriptMaker, you can call an external routine only from a Windows Dynamic Linked Library (DLL), and only if that routine has been written to handle such calls. For example, Windows Application Program Interface (API) routines in USER.EXE, KERNEL.EXE, GDI.EXE, and so on are designed to be called from users scripts. For more information about Windows DLL routines, see the Microsoft Windows Software Development Kit (SDK).

To use an external routine, you use a Declare statement to identify the librarys name and location. The Declare statement must precede the call to the external routine. Like Def and Const statements, the Declare statement must appear outside of any subroutine or function declaration. Declare statements are valid only during the execution of the script.

The API libraries containing the routines are loaded into memory when the routine is called for the first time, not when the script is loaded, so ScriptMaker cannot tell before the call if a script references an external DLL that does not exist.

NOTE: You cannot use the Declare statement to access routines in other ScriptMaker scripts. The Editor allows you to change ScriptMaker scripts to .EXEs, but this does not make them DLLs.

Calling a Function
Calling a subroutine
Declare

External Routines Example

The following example uses the external GetCurrentTime function from USER.EXE. The function returns the number of milliseconds elapsed since Windows was started.

```
'Declare the external function
Declare Function GetCurrentTime Lib "user" As Long

Sub Main( )
   MsgBox Str$(GetCurrentTime)
End Sub
```

FALSE

See Also Example

FALSE is a numeric constant with a value of 0. It is used in relational expression and logical expression.

Conditional Constructs
TRUE

FALSE Example

The following example returns the value FALSE if a specified integer is not odd. Otherwise, the function returns the value TRUE.

```
Function Odd(n As Integer)
   If (n MOD 2) = 1 Then
      Odd = TRUE
   Else
      Odd = FALSE
   End If
End Function
```

FileAttr( )

See Also Example

The FileAttr( ) function returns an integer indicating the mode in which a file was opened or the file handle given to the file by the operating system. It returns the mode in which the file was opened if *attribute* is 1:

- 1, if the file has been opened in input mode
- 2, if the file has been opened in output mode
- 8, if the file has been opened in append mode

It returns the operating system file handle assigned to the file if *attribute* is 2.

Syntax:

FileAttr(*fileNum*, *attribute*)

fileNum	A numeric expression, from 0 to 255, that uniquely identifies a currently open file within your script.
attribute	An integer expression with the value 1 or 2.

FileAttrGet$( )
FileAttrSet
GetAttr
SetAttr

FileAttr( ) Example

The following example calls the FileAttr( ) function to determine the mode in which a file was opened, and then calls it again to get the file handle given to the file by the operating system:

```
Open "testfile" as #1

'theMode should contain the value 8 for append mode
theMode% = FileAttr(1,1)

'fileHandle should now contain the file handle of the file
fileHandle% = FileAttr(1,2)

Close #1
```

FileAttrGet$( )

See Also Example

The FileAttrGet$( ) function returns the attributes of a file as a four-character string expression:

- R as the first character, if the file is read-only; otherwise, a hypen (-)
- A as the second character, if the file has its archive attribute set (indicating it has not been backed up); otherwise, a hypen (-)
- S as the third character, if the file is a system file; otherwise, a hypen (-)
- H as the fourth character, if the file is a hidden file; otherwise, a hypen (-)

~~Syntax:~~PROG_LANG_SYNTAX

FileAttrGet(*filename*)

filename	A string expression containing a complete or relative pathname for a file. It cannot contain wildcards (? or *). If the pathname is not specified, the current directory is searched for the file. An error occurs if the file is not found.

FileAttrSet
FileAttr( )
GetAttr( )
SetAttr(

FileAttrGet$( ) Example

The following example gets the attributes of NDW.EXE and uses the MsgBox statement to display them.

```
attr$ = FileAttrGet$("c:\ndw\ndw.exe")
MsgBox "NDW.EXE attributes: " + attr$
```

FileAttrSet

See Also Example

The FileAttrSet statement changes the attributes of a file.

Syntax:

FileAttrSet *filename, fileAttr*

filename A string expression containing a complete or relative pathname for a file. It cannot contain wildcards (? or *). If the pathname is not specified, the current directory is searched for the file. An error occurs if the file is not found.

fileAttr A string expression of one to four characters specifying the attributes to turn on or off:

Attribute	**Turn On**	**Turn Off**
read-only	R	r
archive	A	a
system	S	s
hidden	H	h

You need only include the characters for the attributes you want to change. The order in which the characters appear in *fileAttr* does not matter.

FileAttrGet$( )
FileAttr( )
GetAttr( )
SetAttr(

FileAttrSet Example

The following example turns off the read-only attribute, turns on the archive attribute, and has no effect on the system and hidden attributes of AUTOEXEC.BAT.

```
FileAttrSet "c:\AUTOEXEC.BAT", "rA"
```

The following example has the identical effect.

```
FileAttrSet "c:\AUTOEXEC.BAT", "Ar"
```

FileCopy

See Also Example

The FileCopy statement copies the specified files to a different drive, a different directory, or a COM or LPT device, with or without first displaying a warning message if an existing file would be overwritten.

Syntax:

FileCopy *from-list, destination* [*, warning*]

from-list A string expression containing complete or relative pathnames for files. The string can contain wildcards (? and *). Multiple filenames or file specifications must be delimited by spaces.

If the pathname is not specified, only the current directory is searched for the files. An error occurs if the files are not found.

destination A string expression containing the target drive, pathname for the directory, pathname for the files, file specification (using the * wildcard only), or COM or LPT device.

If *destination* includes a path but no filename, the new file is given the same name as the source file; if *destination* includes a filename but not a complete pathname, the file is copied to the current directory.

warning A numeric expression: TRUE if a warning dialog box is to appear before overwriting existing files, or FALSE if no warning is to appear. The default value is TRUE.

The warning dialog box displays the name of a file that would be overwritten and gives the user the options of overwriting that file, overwriting that file plus any other existing files, or skipping the file.

FileMove
Kill
Name...As
MkDir
RmDir

FileCopy Example

The following example copies CONFIG.SYS and AUTOEXEC.BAT from the root of the C: drive to the root of the D: drive, but first displays a warning dialog box if the files already exist in the root of the D: drive.

```
FileCopy "c:\config.sys c:\autoexec.bat", "d:"
```

The following example copies all .SYS files in the root of the C: drive to D:\DEVICES. No warning dialog box is displayed.

```
FileCopy "c:\*.sys", "d:\devices\*.sys", FALSE
```

FileDateTime( )

See Also Example

The FileDateTime( ) function returns a date and time as a number of type double in serial format for the first file matching the specification. It is the number of days since December 30, 1899, which is the zero date. A run-time error results if the file does not exist.

Syntax:

FileDateTime(*filename*)

filename A string expression containing a complete or a relative pathname for a file. Wildcards (? or *) can be used. An error occurs if the file does not exist.

Use the date and time functions (for example Hour( ), Minute( ), Second( ), Month( ), Day( ), Year( ), and Weekday( )) to extract the various parts of the time and date from the serial format.

FileTimeGet$( )
FileTimeTouch
CurDir$( )
DiskDrives
DiskFree( )
FileExists( )
FileLen( )
FileParse$( )
FileType( )

FileDateTime( ) Example

The following examples retrieve the date and time of a file:

```
dateAndTime# = FileDateTime("TESTFILE")
fileHour% = Hour(dateAndTime)
fileMinute% = Minute(dateAndTime)
fileSecond% = Second(dateAndTime)
fileMonth% = Month(dateAndTime)
fileDay% = Day(dateAndTime)
fileYear% = Year(dateAndTime)
fileWeekday% = Weekday(dateAndTime)
```

FileDirs

See Also Example

The FileDirs statement fills a string array with the directory names that match a given specification:

Syntax:

FileDirs *dirArray* [, *dirSpec*]

dirArray	The name of the one-dimensional string array that will hold the list of directory names that match *dirSpec* after the statement is executed. This variable can be declared either as a dynamic array, such as Dim a$( ), or as an array with one dimension such as Dim a$(1 to 100). Any other type of string variable causes an error. The statement redimensions the array to hold all of the directory names that match the given specification.
dirSpec	A string expression that specifies the directory. The string can contain wildcards (? and *). If *dirSpec* is not specified, "*.*" is used as the default and finds all the subdirectories in the current directory.

You use the ArrayDims( ) function with *dirArray* to determine if the FileDirs statement found any directories. The ArrayDims( ) function returns 0 if *dirArray* is empty. If the statement finds directories that match the specification, ArrayDims( ) returns the value 1. To find the lowest and highest subscripts for the elements in *dirArray*, and thereby the number of directories found, use the LBound( ) and UBound( ) functions. Even if you declare the array with a specified lower bound, that lower bound is not guaranteed to remain the lower bound if the array has been redimensioned.

NOTE: If the statement find no directories, the LBound( ) or UBound( ) functions cause errors because the array has no elements.

When specifying directories, be aware of the following:

- The function does not return ".." (double period, that is, the parent directory) nor "." (a single period, that is, the current directory) as would appear when doing a "dir" command under DOS.
- If no specification is specified, then the function uses "*.*" as the specification, which returns the directory names in the current directory on the current drive.
- If "." (current directory) is specified as the specification, the function returns only the name of the current directory.
- If the specification does not contain wildcards, only the name of the directory that matches the specification exactly is returned in the array. The contents of that directory are not returned. To get the contents, add "*.*" or a suitable specification to the end of the original specification.
- Using a drive alone as the specification (for example, "C:") or using ".." (parent directory) as the last part of the specification causes an error.
- Using "\" as the last character of the specification causes an error unless it is the first character of the path (for example, "\" or "C:\"), in which case no directory names are returned.
- The specification may contain a complete (for example, "C:\PICTURE\BITMAPS*.BMP") or a relative pathname (for example, "..\LEVEL1*.DGN").

Dir$( )
FileList

FileDirs Example

The first example processes all the directory names in the current directory:

```
'Declare a dynamic string array
Dim directories$( )

'Assume the default specification of *.*
FileDirs directories

'Were any directories found?
If ArrayDims(directories) = 1 Then
   numDirsFound% = UBound(directories) - LBound(directories) + 1
   For i = LBound(directories) To UBound(directories)
      ... 'Do something with each name
   Next i
Else
   'Otherwise, no directories found
   numDirsFound% = 0
End If
```

The next example returns the names of the directories that have a specific extension and are in a specific directory:

```
'Declare a one dimensional string array
Dim dgnNames$(1 To 100)

'Find all *.dgn directories in c:\lvl
FileDirs dgnNames, "c:\lvl\*.dgn"

'Were any directories found?
If ArrayDims(dgnNames) = 1 Then
   numDgnsFound% = UBound(dgnNames) - LBound(dgnNames) + 1
   For i = LBound(dgnNames) To UBound(dgnNames)
      ... 'Do something with *.DGN directory name
   Next i
Else
   'Otherwise, no directories matching spec were found
   numDgnsFound% = 0
End If
```

FileExists( )

See Also Example

To determine whether a file of a specified name exists, use the FileExists( ) function. It returns the value TRUE if the specified name is the name of an existing file. Otherwise, it returns the value FALSE.

Syntax:

FileExists(*filename*)

filename	A string expression containing a complete or a relative pathname for a file. Wildcards (? or *) can be used in *filename*.

CurDir$( )
DiskDrives
DiskFree( )
FileDateTime( )
FileLen( )
FileParse$( )
FileType( )
FileLocate$( )

FileExists( ) Example

Determine whether a file with the name MYFILE exists in the current directory:

```
exist% = FileExists("myfile")
```

Determine whether any files with the extension .SM exist in the root directory:

```
exist% = FileExists("\*.sm")
```

FileLen( )

See Also Example

The FileLen( ) function returns a number of type long indicating the length of the first file that matches the specified filename.

Syntax:

FileLen(*filename*)

filename	A string expression containing a complete or relative pathname for a file. Wildcards (? or *) can be used in *filename*. An error occurs if the file does not exist.

CurDir$( )
DiskDrives
DiskFree( )
FileDateTime( )
FileExists( )
FileParse$( )
FileType( )

FileLen( ) Example

The following example determines the length of the file named TESTFILE:

```
fileLength& = FileLen("TESTFILE")
```

The following example determines the length of the first file in the current directory:

```
fileLength& = FileLen("*.*")
```

FileList

See Also Example

The FileList statement fills a one-dimensional string array with the names of files and directories that match a given specification. The FileList statement also allows you to specify the types of files, such as hidden and system files, to include in the search.

Syntax:

FileList *fileArray* [, *fileSpec* [, *fileAttr*]]

fileArray	The name of the one-dimensional string array that will hold the list of file and directory names that match *fileSpec* after the statement is executed. This variable can be declared either as a dynamic array such as Dim a$( ), or as an array with one dimension such as Dim a$(1 to 100). Any other type of string variable causes an error. The statement redimensions the array to hold all of the directory names that match the given specification.
fileSpec	A string expression that specifies the file and directory names. The string can contain wildcards (? and *). Names of hidden files, system files, directories, and the volume label are ignored. "*.*" is the default. When *fileSpec* is not specified, *fileAttr* cannot be specified.
fileAttr	A numeric expression indicating the sum of the integers representing types of files. The default is 0 (all files regardless of their attributes). The following lists the attributes that can be specified:

ATTR_NORMAL	0	Any file regardless of its attributes
ATTR_READONLY	1	Read-only file
ATTR_HIDDEN	2	Hidden file
ATTR_SYSTEM	4	System file
ATTR_VOLUME	8	Volume label
ATTR_DIRECTORY	16	MS-DOS Directory
ATTR_ARCHIVE	32	File has changed since last backup
ATTR_NONE	64	No attributes are set

To specify a particular combination of the above attributes, use + in the *fileAttr* parameter. For example, to specify files having their read-only, hidden, and system files set (and no other attributes set), use ATTR_READONLY + ATTR_HIDDEN + ATTR_SYSTEM. You can also use a bitwise OR of the desired attributes (for example, ATTR_HIDDEN OR ATTR_SYSTEM, to find files that are either hidden or system). However, ATTR_NORMAL has no effect if used in combination with other constants; for

example, ATTR_NORMAL OR ATTR_READONLY returns only read-only files. Specifying ATTR_NORMAL has the same effect as typing DIR at the DOS prompt.

You use the ArrayDims function with *fileArray* to determine if the FileList statement found any files or directories. The ArrayDims( ) function returns 0 if *fileArray* is empty. If the statement finds directories that match the specification, ArrayDims returns the value 1. To find the lowest and highest subscripts for the elements in *fileArray*, and thereby the number of files and directories found, use the LBound( ) and UBound( ) functions. Even if you declare the array with a specified lower bound, that lower bound is not guaranteed to remain the lower bound if the array has been redimensioned.

NOTE: If the statement finds no files or directories, the LBound( ) or UBound( ) functions cause errors because the array has no elements.

When specifying files, be aware of the following:

- The function returns ".." (double period, that is, the parent directory) and "." (a single period, that is, the current directory) if directories is one of the types of files to be included in the array.
- If no specification is specified, then the function uses "*.*" as the specification, which returns the filenames and directory names in the current directory on the current drive.
- If "." (current directory) is specified as the specification, the function returns only the name of the current directory.
- If the specification does not contain wildcards, only the name of the directory that matches the specification exactly is returned in the array. The contents of that directory are not returned. To get the contents, add "*.*" or a suitable specification to the end of the original specification.
- Using a drive alone as the specification (for example, "C:") or using ".." (parent directory) as the last part of the specification causes an error.
- Using "\" as the last character of the specification causes an error unless it is the first character of the path (for example, "\" or "C:\"), in which case no directory names are returned.
- The specification may contain a complete pathname (for example, "C:\PICTURE\BITMAPS*.BMP") or a relative pathname (for example, "..\LEVEL1*.DGN").

Dir$( )
FileDirs
FileLocate$( )

FileList Example

The following example could be used to process all the filenames in the current directory:

```
'Declare a dynamic string array
Dim files$( )

'Assume the default specification of *.* and ATTR_NORMAL
FileList files

'Were any filenames found?
If ArrayDims(files) = 1 Then
   numFilesFound% = UBound(files) - LBound(files) + 1
   For i = LBound(files) To UBound(files)
      ... 'Do something with each filename
   Next i
Else
   'Otherwise, no filenames found
   numFilesFound% = 0
End If
```

The next example returns the names of the files with a specific extension that are in a specific directory, including read-only, hidden, and archive files:

```
'Declare a one dimensional string array
Dim dgnNames$(1 To 100)

'Find all *.dgn files in c:\lvl,
' including read-only, hidden, and archive

attr% = ATTR_READONLY + ATTR_HIDDEN + ATTR_ARCHIVE
FileList dgnNames, "c:\lvl\*.dgn", attr

'Were any files found?
If ArrayDims(dgnNames) = 1 Then
   numDgnsFound% = UBound(dgnNames) - LBound(dgnNames) + 1
   For i = LBound(dgnNames) To UBound(dgnNames)
      ...          'Do something with *.DGN names
   Next i
Else
   'Otherwise, no filenames matching spec were found
   numDgnsFound% = 0
End If
```

FileLocate$( )

See Also Example

The FileLocate$( ) function returns the complete pathname for the specified file; if the file is not found, a null string is returned.

Syntax:

FileLocate$(*filename*)

filename A string expression containing the complete or relative pathname for the file. The string cannot contain wildcards (* or ?). If the pathname is not specified, the current directory, then the DOS path, are searched for the file.

FileParse$( )
FileExists( )
FileList

FileLocate Example

The following example locates the Norton Desktop application and uses the MsgBox statement to display its complete pathname.

```
ndw$ = FileLocate$ ("ndw.exe")
MsgBox ndw$
```

FileMove

See Also Example

The FileMove statement moves the specified files to a different drive, a different directory, or a COM or LPT device, with or without first displaying a warning message if an existing file would be overwritten.

Syntax:

FileMove *from-list, destination* [, *warning*]

from-list — A string expression containing complete or relative pathnames for files. The string can contain wildcards (? and *). Multiple filenames or file specifications must be delimited by spaces.

If the pathname is not specified, only the current directory is searched for the files. An error occurs if the files are not found.

destination — A string expression containing the target drive, pathname for the directory, pathname for the files, file specification (using the * wildcard only), or COM or LPT device; if *destination* is a COM or LPT device, the files are deleted from the disk drive.

If *destination* includes a path but no filename, the moved file retains its original name; if *destination* includes a filename but not a complete pathname, the file is moved to the current directory.

warning — A numeric expression: TRUE if a warning dialog box is to appear before overwriting existing files, or FALSE if no warning is to appear. The default value is TRUE.

The warning dialog box displays the name of a file that would be overwritten and gives the user the options of overwriting that file, overwriting that file plus any other existing files, or skipping the file.

FileCopy
Kill
Name...As
MkDir
RmDir

FileMove Example

The following example moves CONFIG.BAK and AUTOEXEC.BAK from the root of the C: drive to D:\TMP, but first displays a warning dialog box if the files already exist in D:\TMP.

```
FileMove "c:\config.bak c:\autoexec.bak", "d:\tmp"
```

The following example moves all .BAK files in C:\WORK to D:\BACKUP. No warning dialog box is displayed.

```
FileMove "c:\work\*.bak", "d:\backup\*.bak", FALSE
```

FileParse$( )

See Also Example

To extract a part of a filename such as the path, name, or extension, use the FileParse$( ) function. It returns a string containing the specified part.

Syntax:

FileParse$(*filename* [, *operation*])

filename	A string expression containing a filename to parse. A file by the specified name does not need to exist.
operation	An integer that specifies which portion of the filename to extract. The default is 0 which returns the complete name. The following table gives the return value for each of the possible values:

Value	Description	Example
0	complete name	C:\SHEETS\TEST.DAT
1	drive	C
2	path	C:\SHEETS
3	name	TEST.DAT
4	root	TEST
5	extension	DAT

The filename passed to the function does not need to exist, nor does the path of the directory. However, if a drive is specified in the filename, the specified drive must be a valid drive.

If a complete pathname, which includes the drive letter and the whole directory path, is not included in the filename passed to the function, the function assumes that the filename is relative to the current directory on the current drive. This means that if TEST.DAT is the filename passed to the function, C is the current drive, and \SHEETS is the current directory, then if the complete name (operation value 0) is specified as the operation, C:\SHEETS\TEST.DAT is returned.

CurDir$( )
DiskDrives
DiskFree( )
FileDateTime( )
FileExists( )
FileLen( )
FileLocate$( )
FileType( )

FileParse$( ) Example

The following example returns the complete name of the specified file in the current directory:

```
fullName$ = FileParse$("TESTFILE")
```

The following returns the extension of the specified file:

```
fileExt$ = FileParse$("E:\TESTFILE.TXT",5)       'Should return TXT
```

FileTimeGet$( )

See Also Example

The FileTimeGet$( ) function returns the date and time of the specified file, formatted according to the [intl] section of the WIN.INI file.

Syntax:

FileTimeGet$(*filename*)

filename	A string expression containing the complete or relative pathname for a file. The string cannot contain wildcards (? or *). If the pathname is not specified, the current directory, then the DOS path, are searched for the file. An error occurs if the file is not found.

FileDateTime( )
FileTimeTouch

FileTimeGet$( ) Example

The following example gets the date and time of the AUTOEXEC.BAT file and uses the MsgBox statement to display this information.

```
update$ = FileTimeGet$("c:\autoexec.bat")
MsgBox update$
```

FileTimeTouch

See Also Example

The FileTimeTouch statement assigns the current date and time to one or more files.

Syntax:

FileTimeTouch *filenames*

filenames A string expression containing complete or relative pathnames for files. Multiple filenames must be delimited by spaces. The string cannot contain wildcards (? or *).

If the pathname for a file is not specified, the current directory, then the DOS path, are searched for the file. An error occurs if the file is not found.

FileDateTime( )
FileTimeGet$( )

FileTimeTouch Example

The following example changes the date and time of the AUTOEXEC.BAT and CONFIG.SYS files to the current date and time.

```
FileTimeTouch "c:\AUTOEXEC.BAT c:\CONFIG.SYS"
```

FileType( )

See Also Example

The FileType( ) function return the constant TYPE_DOS if a specified file is a DOS executable file or TYPE_WINDOWS if the file is a Windows executable file.

Syntax:

FileType(*filename*)

filename	A string expression containing a complete or relative pathname for a file. It cannot contain wildcards (? or *). An error occurs if the file does not exist.

If the file is neither a DOS nor Windows executable file, a value other than the above two is returned, and the file type is unknown.

CurDir$( )
DiskDrives
DiskFree( )
FileDateTime( )
FileExists( )
FileLen( )
FileParse$( )

FileType( ) Example

The following example determines the type of the file named SYSINFO.EXE.

```
theType% = FileType("SYSINFO.EXE")
If theType = TYPE_DOS Then
   . . .  'It is a DOS executable file. Do something.
ElseIf theType = TYPE_WINDOWS Then
   . . .  'It is a Windows executable file. Do something.
Else
   . . .  'The file type is unknown. Do something.
End If
```

Fix( )

See Also Example

The Fix( ) function returns the integer part of the specified numeric expression. It retains the sign of the original expression and does not round it off.

Syntax:

Fix(*exprN*)

exprN	A numeric expression in the range for integers.

Abs( )
Int( )
Sgn( )

Fix( ) Example

The following examples illustrate the behavior of the **Fix( )** function.

```
'x is assigned 13
x = Fix(13)

'x is assigned -13
x = Fix(-13)

'x is assigned 13
x = Fix(13.5)

'x is assigned -13
x = Fix(-13.5)
```

For...Next

See Also Example

The For loop allows you to specify an exact number of iterations using a counter that is incremented or decremented by a specified number after each iteration.

The reserved words For and Next mark the beginning and ending of the loop, respectively. Statements between these words execute with each iteration of the loop.

Syntax:

For *counter* = *startNum* **To** *endNum* [**Step** *stepSize*]...
[*statement*]...
Next [*counter*]

counter	The numeric variable that changes from one iteration to the next.
startNum	A numeric expression indicating the starting number for the *Counter*.
endNum	A numeric expression indicating the ending number for the *Counter*. When the value of *counter* is no longer in the range from *startNum* to *endNum*, execution of the loop stops.
stepSize	A numeric expression indicating the number added to the *counter* between iterations of the loop. When a negative number is added to a positive *counter*, the *counter* is decremented. When the Step clause is not included, the default step size is +1.

Do...Loop
Looping Constructs
While...Wend

For...Next Example

The following example uses a **For** loop to request and add scores.

```
Dim Counter As Integer  ' For loop counter.
Dim Score As Integer    ' Input number.
Dim Total As Integer    ' Total of scores.
Dim NumberOfScores As Integer 'Number of scores
...
NumberOfScores = Val(InputBox$ ("How many scores are there?"))
Total = 0
For Counter = 1 To NumberOfScores   ' beginning of loop
   Score = Val(InputBox$ ("Please enter a score."))
   Total = Total + Score
Next        ' end of loop
MsgBox "The total of the scores is " + Str$(Total)
...
```

FreeFile( )

See Also Example

In ScriptMaker, open files are assigned numbers. Instead of keeping track of which file numbers are currently being used in a script and which are available, you can use the FreeFile( ) function to return an available file number. This also lets you use mnemonic names for your files. Save the file number as a variable, because you use that number with every file transaction.

Syntax:

FreeFile[( )]

Open

FreeFile( ) Example

The following example uses the function twice, once to find an available file number for an input file and another time to find an available file number for an output file:

```
'Assign the file number to a variable
inFileNum% = FreeFile
Open "infile" For Input As inFileNum
'Note that the parentheses are optional
outFileNum% = FreeFile( )
Open "outfile" For Output As outFileNum


'Notice that the file numbers were saved in variables
' so that they could be used to close the files
Close inFileNum
Close outFileNum
```

Function...End Function

See Also Example

A function declaration start with the reserved word Function and ends with End Function.

You must *declare* user-defined functions before you can use them. In other words, the declaration of a function must precede the call to that function and be outside of the calling routine. The declaration contains the statements that the call executes.

Syntax:

Function *functionName* [**(** [*parameterList*] **)**] [**As** *type*]
[*localDeclarations*]
[*statements*]
End Function

functionName	Variable name for the function.
parameterList	List of parameters to be passed to the function.
type	Each function is a simple type: Integer, Long, Single, Double, or String. You can identify the type by using the **As** *type* clause of the statement or by adding a type declarator (%, &, or $) to the end of the functions name.
localDeclarations	Declarations of variables to be used in the function.
statements	Other statements to be used in the function.

NOTE: If a 0 or empty string ("") is returned by a function, the function may be missing the assignment statement that gives the functions name a value.

Normally the execution of the function (when it is called) ends with the **End Function** statement. However, you can abort the execution of a routine earlier by including an Exit Function statement in the declaration. For example, if an error occurs, you may want to return to the calling routine without finishing the called routines task.

Function...End Function Examples

The following example declares a string function with no parameters.

```
Function Test( ) As String
   Test = "Hello World"
End Function
```

The next example declares an integer function with an integer parameter.

```
Function Half%(x%)
   Half = x/2
End Function
```

GetAttr( )

See Also Example

The GetAttr( ) function returns an integer indicating the attributes of the first file matching the specification.

Syntax:

GetAttr(*filename*)

filename A string expression containing a complete or relative pathname for a file. Wildcards (? or *) can be used in *filename*. An error occurs if the file does not exist.

The integer returned is a sum of the constants listed below corresponding to the attributes of the file. You can use the AND operator to determine whether a file has particular attributes.

ATTR_NORMAL	0	Normal file
ATTR_READONLY	1	Read-only file
ATTR_HIDDEN	2	Hidden file
ATTR_SYSTEM	4	System file
ATTR_ARCHIVE	32	File has changed since last backup

FileAttr( )
SetAttr
FileAttrSet
FileAttrGet$( )

GetAttr( ) Example

The following example determines whether the AUTOEXEC.BAT file is read-only.

```
If GetAttr("C:\AUTOEXEC.BAT") AND ATTR_READONLY Then
   answer = TRUE         'AUTOEXEC.BAT is read-only
Else
   answer = FALSE        'AUTOEXEC.BAT is not read-only
End If
```

GetCheckBox( )

Overview See Also Example

For the specified check box in the active window or dialog box, the GetCheckBox( ) function returns an integer indicating whether the check box is checked, unchecked, or dimmed:

0 unchecked
1 checked
2 filled (only applicable for three state check boxes)

~~Syntax:~~PROG_LANG_SYNTAX
GetCheckBox(*name* | *ID*)

name A string expression containing the name of the check box.

ID An integer that identifies the check box.

GetComboBoxItem$( )
GetComboBoxItemCount( )
GetEditText$( )
GetListBoxItem$( )
GetListBoxItemCount( )
GetOption( )

GetCheckBox( ) Example

In the following example, if the check box is not checked, then it becomes checked.

```
If GetCheckBox("BOLD") <> 1 Then
   SetCheckBox("BOLD", 1)
End If
```

GetComboBoxItem$( )

Overview See Also Example

For the specified combination box in the active window or dialog box, the GetComboBoxItem$( ) function returns a string containing the text of the specified item.

Syntax:

GetComboBoxItem$({*name* | *ID*}**,** *itemNum*)

name	A string expression containing the name of the combination box. Generally, this is the text in the text control that visually precedes the combination box.
ID	An integer that identifies the combination box.
itemNum	A numeric expression ranging from 1 to the number of lines in the combination box. It is the line number of the selected item.

GetCheckBox( )
GetComboBoxItemCount( )
GetEditText$( )
GetListBoxItem$( )
GetListBoxItemCount( )
GetOption( )

GetComboBoxItem$( ) and GetComboBoxItemCount( ) Example

In the following example, the "File Name:" combination box is searched for an item with the name "foo.doc". If it is found, it is selected.

```
count% = GetComboBoxItemCount("File Name:")
For i = 1 To count
   If GetComboBoxItem$("File Name:", i) = "foo.doc" Then
      SelectComboBoxItem "File Name:", i
      Exit For
   End If
Next
```

GetComboBoxItemCount( )

Overview See Also Example

For the specified combination box in the active window or dialog box, the GetComboBoxItemCount( ) function returns an integer indicating the number of items in the combination box.

Syntax:

GetComboBoxItemCount(*name* | *ID*)

name	A string expression containing the name of the combination box. Generally, this is the text in the text control that visually precedes the combination box.
ID	An integer that identifies the combination box.

GetCheckBox( )
GetComboBoxItem$( )
GetEditText$( )
GetListBoxItem$( )
GetListBoxItemCount( )
GetOption( )

GetEditText$( )

Overview See Also Example

For the specified text box in the active window or dialog box, the GetEditText$( ) function returns a string containing the box's current contents.

Syntax:

GetEditText$(*name* | *ID*)

name	A string expression containing the name of the text box. Generally, this is the text in the text control that visually precedes the text box.
ID	An integer that identifies the text box.

GetCheckBox( )
GetComboBoxItem$( )
GetComboBoxItemCount( )
GetListBoxItem$( )
GetListBoxItemCount( )
GetOption( )

GetEditText$( ) Example

In the following example, a text box labeled "Find What:" has its contents set to "my mistake" if those words are not already in it. The SetEditText statement is not executed unless the text box exists and is enabled.

```
If EditExists("Find What:") = TRUE Then
   If EditEnabled("Find What:") = TRUE Then
      Contents$ = GetEditText$("Find What:")
      If Contents$ <> "my mistake" Then
            SetEditText "Find What:", "my mistake"
      End If
   End If
End If
```

GetListBoxItem$( )

Overview See Also Example

For the specified list box in the active window or dialog box, the GetListBoxItem$( ) function returns a string containing the text of the specified item.

Syntax:

GetListBoxItem$({*name* | *ID*}**,** *itemNum*)

name	A string expression containing the name of the list box. Generally, this is the text in the text control that visually precedes the list box.
ID	An integer that identifies the list box.
itemNum	A numeric expression ranging from 1 to the number of lines in the list box. It is the line number of the selected item.

GetCheckBox( )
GetComboBoxItem$( )
GetComboBoxItemCount( )
GetEditText$( )
GetListBoxItemCount( )
GetOption( )

GetListBoxItem$( ) and GetListBoxItemCount( ) Example

In the following example, the "File Name:" list box is searched for an item with the name "foo.doc". If it is found, it is selected.

```
count% = GetListBoxItemCount("File Name:")
For i = 1 To count
   If GetListBoxItem$("File Name:", i) = "foo.doc" Then
      SelectListBoxItem "File Name:", i
      Exit For
   End If
Next
```

GetListBoxItemCount( )

Overview See Also Example

For the specified list box in the active window or dialog box, the GetListBoxItemCount( ) function returns an integer indicating the number of items in the list box.

Syntax:

GetListBoxItemCount(*name* | *ID*)

name	A string expression containing the name of the list box. Generally, this is the text in the text control that visually precedes the list box.
ID	An integer that identifies the list box.

GetCheckBox( )
GetComboBoxItem$( )
GetComboBoxItemCount( )
GetEditText$( )
GetListBoxItem$( )
GetOption( )

GetOption( )

Overview See Also Example

For the specified option button in the active window or dialog box, the GetOption( ) function returns TRUE if the option is set or FALSE if the option is not set.

Syntax:

GetOption(*name* | *ID*)

name	A string expression containing the name of the option button.
ID	An integer that identifies the option button.

GetCheckBox( )
GetComboBoxItem$( )
GetComboBoxItemCount( )
GetEditText$( )
GetListBoxItem$( )
GetListBoxItemCount( )

GetOption( ) Example

The following example selects the option button "Show All" if it is not already selected.

```
If GetOption("Show All") <> TRUE Then
   SetOption "Show All"
End If
```

GoSub...Return

See Also Example

The GoSub statement is supplied for compatibility with the underlying BASIC language, but GoSub can easily be replaced by a user-defined functions and subroutines that is much more powerful. For example, you can pass parameters to a subroutine but not to a label.

The GoSub statement uses the label as a starting point and the Return statement as an ending point. The execution of the Return statement transfers control to the statement following the GoSub statement.

The GoSub statement must be in the same subroutine or function as the label to which it transfers control. You put the sequence of statements that starts with the label and ends with Return at the end of the subroutine or function. The Exit Sub or ~~Exit Function~~ statement, which terminates the routine, must precede the label or its statements are executed again at the end of the subroutine or function. If you have several *label*...Return sequences, they appear one after the other between the Exit Sub or Exit Function statement and the End Sub statement.

Syntax:

Sub Main
...
GoSub *label*
...
Exit {**Sub** | **Function**}

label:
[*statement*]...
Return
End Sub

label	An identifier used to indicate the starting point of a GoSub routine.
statement	The statements executed by the GoSub routine.

NOTE: The GoSub statement can be used without the Return statement. In that case, it acts exactly as the GoTo statement and is not recommended.

Control Constructs
GoTo
Labels

GoSub...Return Example

The following example uses the **GoSub** statement to repeat the same sequence of statements throughout a script.

```
Sub Main
...
'Write standard header to first file
GoSub PrepareHeader
...
'Write standard header to second file
GoSub PrepareHeader
...
'Write standard header to third file
GoSub PrepareHeader
...
'The Exit Sub keeps the script from executing
'the PrepareHeader routine unless it is send
'to it
Exit Sub

PrepareHeader:
   'sequence of statements that write header lines to a file

Return
End Sub
```

GoTo

See Also Example

You should avoid using GoTo statements in ScriptMaker. Using several GoTo statements can transfer control from label to label to label convoluting the order of the statements and making a script difficult to understand. The term *spaghetti code* refers to scripts that misuse GoTo statements in this way. The GoTo statement must be in the same subroutine or function as the label to which it transfers control.

Syntax:

GoTo *label*
...
label:
[*statement*]...

label	An identifier used to indicate the starting point of a GoSub routine.
statement	The statements executed by the GoSub routine.

NOTE: The On Error GoTo statement, is not the same as the **GoTo** statement explained here, and does not need to be avoided like the **GoTo** statement.

GoTo Example

The following example uses a GoTo statement to skip the sequence of statements between the If statement and LabelOne.

```
...
If JobDone Then
   GoTo LabelOne
End If
... 'sequence of statements
LabelOne:
... rest of statements in subroutine
```

GroupBox

Overview See Also Example

The GroupBox statement defines a group box within a dialog box template. A group box is a visual element used to enclose other controls within a dialog box.

Syntax:

GroupBox *x, y, width, height*, *name*

x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in dialog units. The upper-left corner of the window is 0, 0.
width, *height*	The integer expressions indicating the width and height of the dialog box in dialog units.
name	String variable or literal containing the name of the group box. It can contain an ampersand & in front of the character to be used as an accelerator key.

Begin Dialog...End Dialog
Dialog
Dialog( )

Hex$( )

See Also Example

The Hex$( ) function rounds a specified decimal number to the nearest whole number and then converts it to its hexadecimal, base 16, equivalent. The function returns a string containing the hexadecimal equivalent of the specified numeric expression. Each character of the string is a digit of the hexadecimal number. It returns up to four hex digits for an integer, and up to eight for a long.

Syntax:

Hex$(*exprN*)

exprN	A numeric expression in the range for longs.

Asc( )
Chr$( )
Oct$( )
Str$( )
Val( )

Hex$( ) Example

The following example converts the decimal number 16 to hexadecimal.

```
hexOf16$ = Hex$(16)      'Result should be the string "10"
```

HLine

See Also Example

HLine scrolls left or right a specified number of columns in an active window's viewport.

During a recording session with the Recorder, clicking the arrow button at either end of a horizontal scroll bar generates an HLine statement.

Syntax:

HLine [*columns*]

lines Number of columns to scroll. If positive, the scrolling is to the right. If negative, the scrolling is to the left. The default is to scroll right one column.

AppActivate
HPage
HScroll
VLine
VPage
VScroll
WinActivate

HLine Example

The following example scrolls to the right 10 lines using the horizontal scroll bar.

```
HLine 10
```

Hour( )

See Also Example

The Hour( ) function returns a number in the range from 0 to 23 representing the hour of the serial time.

Syntax:

Hour(*serialDateTime*)

serialDateTime	Serial time, a number of type double, from which the hour is to be extracted.

DateSerial( )
DateValue( )
Day( )
Minute( )
Month( )
Now( )
Second( )
TimeSerial( )
TimeValue( )
Weekday( )
Year( )

HPage

See Also Example

HPage scrolls left or right by a specified number of pages in an active window's viewport. During a recording session with the Recorder, clicking in the scroll area on either side of the scroll box generates an HPage statement.

Syntax:

HPage [*pages*]

pages Number of pages to scroll. If positive, the scrolling is to the right. If negative, the scrolling is to the left. The default is to scroll right one page.

AppActivate
HLine
HScroll
VLine
VPage
VScroll
WinActivate

HPage Example

The following example scrolls to the left two pages using the horizontal scroll bar.

```
HPage -2
```

HScroll

See Also Example

The HScroll statement positions the scroll box a percentage of the way across the total range of a horizontal scroll bar in the active window's viewport. During a recording session with the Recorder, dragging the scroll box to a new position within the scroll bar generates an HScroll statement.

Syntax:

HScroll *percentage*

percentage	An integer specifying a percentage of the scroll bar, and, therefore, the location at which to place the scroll box.

AppActivate
HLine
HPage
VLine
VPage
VScroll
WinActivate

HScroll Example

The following example sets the horizontal scroll box in the middle of the scroll bar.

```
HScroll 50
```

IconArrange

See Also Example

The IconArrange statement aligns minimized application, drive, and group windows along the bottom of the desktop. This statement has the same effect as choosing Arrange Desktop Icons from the Window menu of Norton Desktop, or clicking the Arrange Icons command button on the Windows Task List. The IconSpacing and IconVerticalSpacing settings in the WIN.INI file, [Desktop] section, determine the amount of space between icons.

Syntax:

IconArrange

DesktopSetColors
DesktopSetWallpaper
DesktopTile
DesktopCascade

IconArrange Example

The following example executes IconArrange if the user clicks OK in response to the message displayed by the AnswerBox( ) function.

```
message$ = "Click OK to arrange the icons on your desktop now"
response = AnswerBox(message)
If response = 1 Then IconArrange
```

If...Then...Else...End If

See Also Example

The If statement uses logical expressions (also called Boolean expressions) which evaluate as either true or false, to choose which sequence of statements to execute. The sequence associated with a true expression is executed.

The simplest form of the If statement executes a sequence of statements when the expression is true and bypasses that sequence when the expression is false. It starts with the reserved word If followed by a logical expression and the reserved word Then. The Then is followed by the sequence of statements. The end of the sequence is indicated by the closing End If.

Syntax:

If *exprL* **Then**
    [*statement*]...
End If

Or, if the sequence consists of only one statement:

If exprL **Then** [statement]

exprL	A logical expression. A logicial expression contains relational and/or logical operators.
statement	An executable statement.

If you want to execute another sequence of statements when the expression is not true, use Else:

Syntax:

If *exprL* **Then**
    [*statement*]...
[**Else**
    [*statement*]...]
End If

Or, if each sequence consists of only one statement:

If *exprL* **Then**[*statement*][**Else** [*statement*]]

If three or more expressions are to be evaluated, use ElseIf. Each If statement can have only one Else. It can have as many ElseIfs as you want.

Syntax:

If *exprL* **Then**
    [*statement*]...
[**ElseIf** *exprL* **Then**
    [*statement*]...]...
[**Else**
    [*statement*]...]
End If

Conditional Constructs
Select Case...End Select

If...Then...Else...End If Examples

In the following If...Then example, the Total appears on the screen only when it is greater than zero.

```
Dim Total As Integer
...
If Total > 0 Then
   MsgBox "TOTAL: " + Str$(Total)
End If
...
```

In this If...Then...Else example, the computer displays one message when the Total is greater than zero and another when it is not.

```
Total% = 0
...
If Total > 0 Then
   MsgBox "TOTAL:  " + Str$(Total)
Else
   MsgBox "Total is 0."
   ...
End If
...
```

In this If...Then...ElseIf example, the series of ElseIf statements include all the possible values for Choice.

```
...
If (Choice  = 1) and (Count < Total) Then
   Count = Count + 1
ElseIf (Choice = 2) Then
   Count = Count - 1
ElseIf (Choice = 3) Then
   Count = 0
   MsgBox "Starting over."
Else
   MsgBox "Invalid choice."
End If
...
```

Input

See Also Example

The Input statement reads data from a file that has been opened in input mode. The data items of the input file are read into the input variables in consecutive order.

Syntax:

Input [#] *fileNum*, *variable* [, *variable*] ...

fileNum	A numeric expression, from 0 to 255, that uniquely identifies a currently open file within your script.
variable	Name of the variable to receive data from the file. The type of the variable must match the data.

Files read by Input statements must have the following properties:

- Data items on the same line must be separated by commas.
- A carriage-return/linefeed can also be used to separate data items. This means that a single Input statement can be used to read across multiple lines.
- String data items must be enclosed within quotes, for example, "Hello World".
- The variable that will store an item must be the same type as the item. For example, a numeric variable cannot be used to read a string.

Input statements easily read files that were created using Write statements. The Input statement reads items separated by commas, while the Write statement uses commas to separate the items it writes.

Input$( )
Line Input #
Open
Print #
Seek
Write #

Input # Example

TESTFILE contains the following ten lines, each of which contains one of the first ten positive integers and its square:

```
1,1
2,4
3,9
4,16
5,25
6,36
7,49
8,64
9,81
10,100
```

The following examples read the integers into the array named N and the squares into the array named NSQUARED:

```
Dim N(1 To 10) As Integer
Dim NSQUARED(1 To 10) As Integer

Open "testfile" For Input As #4
For i = 1 To 10
   Input #4, N(i), NSQUARED(i)
Next i
```

This time TESTFILE contains the following two lines:

```
5,"Hello"
6,"World!"
```

The following examples read the items into the corresponding variables:

```
Open "testfile" For Input As #99

'Note that this also reads the 6 on the 2nd line
Input #99, five%, hello$, six%

'Now read in the last string
Input #99, world$
```

Input$( )

See Also Example

The Input$( ) function reads a specified number of characters from the file that has been opened in input mode. The function reads all characters, including spaces and carriage-returns and returns a string containing those characters.

Syntax:

Input$(*charNum*, [#]*fileNum*)

charNum	A numeric expression that specifies the number of characters to read from the file.
fileNum	A numeric expression, from 0 to 255, that uniquely identifies a currently open file within your script.

Input #
Line Input #
Open
Print #
Seek
Write #

Input$( ) Example

The following example uses **Input$( )** to read the first ten characters of a file into the string variable BUFFER:

```
Open "testfile" For Input As #1
BUFFER$ = Input$(10,#1)
```

InputBox$( )

Overview See Also Example

The InputBox$( ) function allows you to display a predefined dialog box that contains:

- A message that you specify.
- A text box for a response from the user that can contain a maximum of 255 characters.
- A name for the dialog box that you specify.
- The OK and Cancel command buttons.

The function returns the contents of the text box if the user clicks OK or an empty string if the user cancels the dialog box.

You can position the dialog box in the current window, or use the default position, which centers the dialog box in the current window. You can display a default string expression in the text box for the user.

The dialog box does not resize itself to fit the message. It accommodates only 12 lines of text with about 20 characters per line.

Syntax:

InputBox$ (*message* [, *name* [, *contents* [, *x* , *y*]]])

message	A string expression that the user must respond to.
name	A string expression for the name of the dialog box. By default, there is no name.
contents	A string expression used as the initial contents of the text box. The user can accept this or type a new string. The default is an empty string.
x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in twips. The upper-left corner of the window is 0, 0. By default, the dialog box is centered in the window.

AskBox$( )
AskPassword$( )

InputBox$( ) Example

The following example fills the text box with a name of a company: ACME. If ACME is the company most often making requests, making its name the default saves the user time and does not stop the user for changing the name when appropriate.

```
theMessage$ = "What company makes the request?"
theTitle$ = "Requesting Company"
Company$ = InputBox$(theMessage, theTitle, "ACME")
```

InStr( )

See Also Example

The InStr( ) function finds the position of the first occurrence of a substring within a string. It returns a number indicating the starting position of the first occurrence of *subStr* within *searchStr*. If the substring could not be found within the search string, then 0 is returned.

Syntax:

InStr([*startPos*,] *searchStr*, *subStr*)

startPos	A numeric expression giving the starting position within the search string. The first position, which is also the default starting position, of the search string is position 1.
searchStr	A string expression indicating the string to search.
subStr	A string expression indicating the substring to find.

Item$( )
Line$( )
Word$( )

InStr( ) Example

The following example deletes trailing percent signs from LastName:

```
'Find the first occurrence of the trailing percent signs
Position% = InStr(LastName, "%")
If Position > 1 Then
   LastName = Left$(LastName, Position - 1)
End If
```

The following example specifies a starting position because the substring is known to occur near the end of the search string.

```
'This will be the string to find
FindStr$ = "Find Me"

'Add 15000 a's to the beginning of the search string
SearchStr$ = String$(15000,"a") + FindStr

'Start searching from position 10000, result should be 15001
Position% = InStr(10000, SearchStr, "Find Me")
```

Int( )

See Also Example

The Int( ) function returns the first integer less than the specified number. The sign is preserved.

Syntax:

Int(*exprN*)

exprN	A numeric expression in the range for integers..

Abs( )
Fix( )
Sgn( )

Int( ) Example

The following examples illustrate the behavior of the **Int( )** function.

```
'x is assigned 13 because 13 < 13.7
x = Int(13.7)

'x is assigned -14 because -14 < -13.2
x = Int(-13.2)
```

Item$( )

See Also Example

The Item$( ) function parses, or in other words, extracts text items from text. The items are separated by specified delimiters. It returns a string containing all the items starting with the first one specified and ending with the last one specified.

Syntax:

Item$(*text*, *first*[, *last*[, *delimiters*]])

text	A string expression indicating the text to parse.
first	A numeric expression specifying the first item to retrieve. Item 1 is the first item of the text.
last	A numeric expression specifying the last item to retrieve. The default is the value of *first* so only one item is retrieved at a time.
delimiters	A string expression specifying the characters to use as delimiters. Each character in the expression is considered a delimiter. The default delimiters are commas and carriage-returns/linefeeds. To specify *delimiters*, *last* must have also been specified.

If the number of items to retrieve is greater than one, then all the delimiters separating the retrieved items are also included in the returned string.

If *first* is greater than the number of items in *text*, an empty string is returned.

If *last* is greater than the number of items in *text*, then all the items from *first* to the end of *text* are returned.

ItemCount( )
Line$( )
LineCount( )
Word$( )
WordCount( )

Item$( ) and ItemCount( ) Example

In the following example, a list of names is stored in the string variable itemText. A For loop places each name from the string into a string array. No delimiters are specified in either the call to the ItemCount( ) function or the call to the Item$( ) function. The comma is one of the default delimiters, so it does not have to be specified.

```
Dim nameList$(10)

'The text with the names
itemText$ = "John,Mary Jane,Ken,T.S."

For i=1 To ItemCount(itemText)
   'Parse each name
   nameList(i) = Item$(itemText, i)
Next i
```

In the next example, a list of numbers is stored in the string variable itemText. A For loop extracts each of the numbers, which are separated from one another by colons. The calls to ItemCount( ) and to Item$( ) specify the colon as the delimiter.

```
Dim itemList$(10)

itemText$ = "123:456:789:0"

For i=1 To ItemCount(itemText, ":")
   itemList(i) = Item$(itemText, i, i, ":")
Next i
```

ItemCount( )

See Also Example

The ItemCount( ) function returns an integer indicating the number of items in the specified text. Items are separated by the specified delimiters.

Syntax:

ItemCount(*text*[, *delimiters*])

text	A string expression containing the text to parse.
delimiters	A string expression specifying the characters to use as delimiters. The default delimiters are commas and carriage-returns/linefeeds.

Item$( )
Line$( )
LineCount( )
Word$( )
WordCount( )

Keystroke Specification Format

See Also

DoKeys, QueKeys, QueKeyUp, QueKeyDn, and SendKeys all use the same string format for specifying keystrokes:

- To specify any printable character from the keyboard, just use that key (for example, "h" for lowercase h, and "H" for uppercase h).
- To specify a sequence of keystrokes, just append keystrokes, one after the other, in the order desired (for example, "asdf" or "dir /p").
- The plus sign (+), caret (^), tilde (~), percent sign (%), parentheses, square brackets, and curly braces are used to specify keystroke combinations. For example "^d" indicates Ctrl+D. These special uses appear later in this list. To specify one of these characters as itself, a single or shifted keystroke with no special meaning, enclose the corresponding character within curly braces. For example, "{(}" specifies a left parenthesis, or "{%}" to specify the percent symbol).
- To specify keys that are not displayable characters, enclose the description of the key within curly braces For example, {ENTER} is the Enter key and {UP} is the UpArrow key). A list of these keys follows:

{BACKSPACE}	{BS}	{BREAK}	{CAPSLOCK}
{CLEAR}	{DELETE}	{DEL}	{DOWN}
{END}	{ENTER}	{ESCAPE}	{ESC}
{HELP}	{HOME}	{INSERT}	{LEFT}
{NUMLOCK}	{NUMPAD0}	{NUMPAD1}	{NUMPAD2}
{NUMPAD3}	{NUMPAD4}	{NUMPAD5}	{NUMPAD6}
{NUMPAD7}	{NUMPAD8}	{NUMPAD9}	{NUMPAD/}
{NUMPAD*}	{NUMPAD-}	{NUMPAD+}	{NUMPAD.}
{PGDN}	{PGUP}	{PRTSC}	{RIGHT}
{TAB}	{UP}	{F1}	{SCROLLLOCK}
{F2}	{F3}	{F4}	{F5}
{F6}	{F7}	{F8}	{F9}
{F10}	{F11}	{F12}	{F13}
{F14}	{F15}	{F16}	

- To specify keystrokes combined with a modifier key, such as Shift, Ctrl, or Alt, precede the keystroke specification with "+", "^", or "%" respectively. For example, "+{ENTER}" means Shift+Enter, "^c" means Ctrl+C, "%{F2}" means Alt+F2).
- To specify a modifier key combined with a sequence of consecutive keys, group the key sequence within parentheses and precede it with either "+", "^", or "%" (for example, "+{abc}" means the Shift key is held down while the a, b, and c keys are typed in consecutive order, "^({F1}{F2})" means the Ctrl key is held down while the F1 and then the F2 keystrokes are specified).
- The "~" can be used as a shortcut for embedding the ENTER keystroke within a key sequence. For example, "ab~de" means the Enter key was pressed after "ab".
- To embed quotes, use two quotes in a row, for example, "This is a ""test"" of the system".
- To repeat a keystroke, enclose the keystroke and a repeat count within curly braces (for example, "{a 10}" means "Produce 10 "a" keystrokes"; "{ENTER 2}" means "Produce two ENTER keystrokes").

DoKeys, QueKeys
QueKeyUp, QueKeyDn
SendKeys

Kill

See Also Example

The Kill statement deletes the specified files from disk in the same way as the DOS DEL command.

Syntax:

Kill *fileSpec*

fileSpec A string expression containing a complete or relative pathname. The string can contain wildcards (* and ?). All files matching the string are deleted.

ChDir
ChDrive
MkDir
Name...As
RmDir
FileCopy
FileMove

Kill Example

The following example deletes all files in the current directory:

```
Kill "*.*"  'Be careful when doing this
```

The following example deletes the file TESTFILE at the root of the C: drive:

```
Kill "C:\TESTFILE"
```

Labels

See Also

ScriptMaker allows you to use labels with either GoTo or GoSub statements, but it is not recommended. These statements affect the flow of execution by transferring control to the label and the statements that follow it. The GoTo and GoSub statements must be in the same subroutine or function as the labels to which they transfer control. Each label consists of an identifier followed by a colon.

The On Error GoTo statement also uses a label. The label is used to indicate where control should be transferred when a run-time error occurs.

Syntax:

label:

label	A valid identifier used to mark the statement to which control is transferred.

GoSub...Return
GoTo

LBound( )

See Also Example

The LBound( ) function returns an integer indicating the lowest subscript number for the specified dimension of the specified array.

Syntax:

LBound(*arrayName* [, *dimension*])

arrayName	The name of the array.
dimension	The dimension. The default is the first dimension.

NOTE: If the LBound( ) function is used on an array with no dimensions, then a run-time error will occur. The ArrayDims( ) function can be used to first check if the array has any dimensions.

ArrayDims( )
ArraySort
Dim
Option Base
ReDim
UBound( )

LBound( ) Example

The following example finds the lower bound for subscripts in the first dimension of a two-dimensional array using the **LBound( )** function.

```
Dim Array1(0 To 3, 0 To 2) As Integer

'Determine the lower bound
lowest_subscript = LBound(Array1)
```

LCase$( )

See Also Example

The LCase$( ) function converts the uppercase letters in a string to lowercase. It returns a string containing all the characters of the specified string in lowercase.

Syntax:

LCase$(*exprS*)

exprS	A string expression to be converted to lowercase.

UCase$( )

LCase$( ) Example

The following call to LCase$( ) should result in the string "this is only a test" being assigned to the variable newString.

```
newString$ = LCase$("This is Only a Test!")
```

Left$( )

See Also Example

The Left$( ) function returns a string containing the leftmost *n* characters from the specified string. If *n* is greater than or equal to the number of characters in *exprS*, the entire string is returned.

Syntax:

Left$(*exprS, n)*

exprS	A string expression from which to retrieve characters.
n	The number of characters to retrieve.

LTrim$( )
Mid$( )
Right$( )
RTrim$( )

Left$( ) Example

The following example deletes trailing percent signs from LastName:

```
'Find the first occurrence of the trailing percent signs
Position% = InStr(LastName, "%")
If Position > 1 Then
   'Retain only the leftmost characters
   LastName = Left$(LastName, Position - 1)
End If
```

Len( )

See Also Example

The Len( ) function returns the number of characters in the specified string.

Syntax:

Len(*exprS)*

exprS	A string expression whose length is to be determined.

ItemCount( )
LineCount( )
WordCount( )

Len( ) Example

The following example uses the **Len( )** function to determine the length of a message:

```
Length = Len(Message)
```

Let

See Also Example

The reserved word Let optionally precedes an assignment statement. An assignment statement places the value to the right of the assignment operator (=) in the memory location represented by the variable to the left of the operator.

Syntax:

[**Let**] *var* **=** *expr*

The variable *var* is assigned the value of the expression *expr*.

The initial value of a variable is assigned to the variable when it is declared. Zero is the initial value of a numeric variable, and an empty string is the initial value of a string variable. An assignment statement changes that value.

NOTE: The equal sign is also used as a relational operator which compares two quantities to see if they are equal. The difference is that the assignment operator gives a variable a value, and a comparison for equality returns a value of TRUE (if equal) or FALSE (if unequal).

Assignment Statements

Let Example

Both of the following examples assign the value 5 to x.

```
Let x = 5
```

```
x = 5
```

You can use a variable on both sides of the first assignment statement that uses it. For example, the following example increases the value of the variable Counter by one.

```
Counter = Counter + 1
```

When this statement is executed, the value of the Counter on the right side is 0, its initial value, and the value of the Counter on the left is the sum of 0+1, which is 1.

Line Input

See Also Example

The Line Input statement reads one line from a file (opened in input mode) into the specified string variable. After the line has been read, the file pointer is advanced to the beginning of the next line.

Syntax:

Line Input [#]*fileNum*, *text*

fileNum	A numeric expression, from 0 to 255, that uniquely identifies a currently open file within your script.
text	A string variable that will contain the line of text read from the file.

Input #
Input$( )
Open
Print #
Seek
Write #

Line Input # Example

The following example can be used to skip past the first ten lines of a file:

```
Open "testfile" For Input As #7
For i = 1 To 10
   Line Input #7, dontcare$
Next i
```

Line$( )

See Also Example

The Line$( ) function returns a string containing the lines of text starting with the first line specified and ending with the last line specified. Lines are delimited by carriage-return/linefeeds.

Syntax:

Line$(*text*, *first*[, *last*])

text A string expression containing the text to parse.

first A numeric expression specifying the first line to retrieve. Line 1 is the first line of the text.

last A numeric expression specifying the last line to retrieve. The default is 1 so one line is returned.

If the number of lines to retrieve is greater than one, then all the carriage-return/linefeeds separating the retrieved lines are also included in the returned string.

If *first* is greater than the number of lines in *text*, an empty string is returned.

If *last* is greater than the number of lines in *text*, then all the lines from *first* to the end of *text* are returned.

Item$( )
ItemCount( )
LineCount( )
Word$( )
WordCount( )

Line$( ) and LineCount( ) Example

In the following example, a string is assigned an address. Each component of the address is on a separate line. The Line$( ) function is then used to extract the name, street, city, and other information from it. LineCount( ) is used to determine whether there is any other information besides the name, street, and city.

```
'Carriage-return/linefeed pair
crlf$ = Chr$(13) + Chr$(10)

'Make an address for the example
address$ = "John Doe" + crlf$
address = address + "123 Old Lane" + crlf$
address = address + "New City" + crlf$
address = address + "Other Information"

'Now parse the address
name$ = Line$(address, 1)
street$ = Line$(address, 2)
city$ = Line$(address, 3)

'Is there other information
If LineCount(address) > 3 Then
   other$ = Line$(address, 4, LineCount(address))
End If
```

LineCount( )

See Also Example

The LineCount( ) function returns an integer indicating the number of lines in the specified text. Lines are delimited by carriage-return/linefeed pairs.

Syntax:

LineCount(*text*)

text A string expression containing the text to parse.

Item$( )
ItemCount( )
Line$( )
Word$( )
WordCount( )

ListBox

Overview See Also Example

The ListBox statement defines a list box that appears within a dialog box template.

Syntax:

ListBox *x, y, width, height*, *itemsArray*, *.field*

x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in dialog units. The upper-left corner of the window is 0, 0.
width, *height*	The integer expressions indicating the width and height of the dialog box in dialog units.
itemsArray	A one-dimensional string array that contains the elements to be placed into the list box.
.field	An integer variable used to set and/or retrieve subscript of the array element selected from the list box. Setting this field to a subscript from *itemsArray* gives the list box an initial selection.

Begin Dialog...End Dialog
Dialog
Dialog( )

ListBoxEnabled( )

Overview See Also Example

ListBoxEnabled( ) determines whether the list box with the specified name or ID is enabled in the active window or dialog box. This allows you to avoid the run-time error that occurs if a statement is executed for a list box that is disabled (dimmed). The function returns TRUE if the list box is enabled and FALSE if the list box is dimmed. If the list box does not exist in the current dialog box, a run-time error occurs.

Syntax:

ListBoxEnabled(*name* | *ID*)

name	A string expression containing the name of the list box. Generally, this is the text in the text control that visually precedes the list box.
ID	An integer that identifies the list box.

ListBoxExists( )
SelectListBoxItem
GetListBoxItem$( )
GetListBoxItemCount( )
ButtonEnabled( )
CheckBoxEnabled( )
ComboBoxEnabled( )
EditEnabled( )
OptionEnabled( )

ListBoxExists( ), ListBoxEnabled( ), and Select ListBox Example

The following example checks if the list box named "Files:" both exists and is enabled before it selects an item from the list box.

```
If ListBoxExists("Files:") = TRUE Then
   If ListBoxEnabled("Files:") = TRUE Then
      SelectListBoxItem "Files:", "AUTOEXEC.BAT"
   End If
End If
```

The following is a simple recorded example in which an application is activated, then an item is selected from a list box:

```
'Select the Box Types application
'Make the Box Types application active
WinActivate "Box Types"
'Select the "Big" item in the "Box Size" list box
SelectListBoxItem "Box Size", "Big"
```

ListBoxExists( )

Overview See Also Example

ListBoxExists( ) checks for the existence of a list box with the specified name or ID in the active window or dialog box. This allows you to avoid the run-time error that occurs if a statement is applied to a list box that does not exist. The function returns TRUE if the list box exists and FALSE otherwise.

Syntax:

ListBoxExists(*name* | *ID*)

name	A string expression containing the name of the list box. Generally, this is the text in the text control that visually precedes the list box.
ID	An integer that identifies the list box.

ListBoxEnabled( )
SelectListBoxItem
GetListBoxItem$( )
GetListBoxItemCount( )
ButtonExists( )
CheckBoxExists( )
ComboBoxExists( )
EditExists( )
OptionExists( )

Loc( )

See Also Example

The Loc( ) function returns a number in the range from 0 to 2,147,483,647, which gives the current position of the file pointer. The first character of the file, which is also the first character of the first line, is at position 0.

Syntax:

Loc(*fileNum*)

fileNum	A numeric expression, from 0 to 255, that uniquely identifies a currently open file within your script.

EOF( )
FileAttr( )
LOF( )
Open
Seek
Seek( )

LOF( )

See Also Example

The length of a file is easily determined with the use of the LOF( ) function, which returns an long indicating the number of bytes in the specified file.

Syntax:

LOF(*fileNum*)

fileNum	A numeric expression, from 0 to 255, that uniquely identifies a currently open file within your script.

NOTE: You can determine the length of the file without having to open it by using FileLen( ).

EOF( )
FileAttr( )
Loc( )
Open
Seek
Seek( )

LOF( ) Example

In the following example, a file is opened, and then its length is stored in a variable:

```
Open "testfile" As #1
fileLength = LOF(1)
Close #1
```

Log( )

See Also Example

The Log( ) function returns the natural logarithm of the specified number as a number of type double. The natural logarithm is the power to which a fixed number, the base *e* (2.71828), must be raised to produce the number, which cannot be zero.

Syntax:

Log(*exprN*)

exprN	A numeric expression for which to calculate the natural logarithm.

Exp( )

Log( ) Example

The following example shows a simple use of the **Log( )** function.

```
e# = 2.71828       'Definition of e
lne# = Log(e)      'Natural logarithm of e should equal 1
```

Logical Expressions

See Also Example

A logical expression is any expression that uses relational or logical operators and evaluates to either true or false. The predefined constants TRUE and FALSE, which have the numeric values -1 and 0 respectively, can be used in logical expressions.

ScriptMaker performs operations in logical expressions according to an order of precedence. Higher-priority operations are evaluated first. The following table shows the operators that can appear in logical expressions from highest to lowest priority.

Operators in Order of Precedence		Meaning
Parentheses	**( )**	Parentheses. Logically groups expressions.
Arithmetic Operators	^	Exponentiation. Raises a number to a power: 3 squared is 3^2.
	-	Unary minus changes the sign of a number.
	* /	Multiplication and division.
	\	Integer division.
	MOD	Modulo (*exprN1* **MOD** *exprN2* results in the remainder of *exprN1* **** *exprN2*).
	+ -	Addition and subtraction. The plus also concatenates strings; the minus also gets the number of seconds between two times and the days between two dates.
Relational Operators	=	Equal to.
	<>	Not equal to.
	<	Less than.
	<=	Less than or equal to.
	>	Greater than.
	>=	Greater than or equal to.
Logical Operators	NOT	Evaluates to **true** when the Boolean expression is **false**, and vice versa.
	AND	*TrueExpr* **AND** *TrueExpr* equals **true**. *TrueExpr* **AND** *FalseExpr* equals **false**. *FalseExpr* **AND** *TrueExpr* equals **false**. *FalseExpr* **AND** *FalseExpr* equals **false**.
	OR	*TrueExpr* **OR** *TrueExpr* equals **true**. *TrueExpr* **OR** *FalseExpr* equals **true**. *FalseExpr* **OR** *TrueExpr* equals **true**. *FalseExpr* **OR** *FalseExpr* equals **false**.
	XOR	*TrueExpr* **XOR** *TrueExpr* equals **false**. *TrueExpr* **XOR** *FalseExpr* equals **true**. *FalseExpr* **XOR** *TrueExpr* equals **true**. *FalseExpr* **XOR** *FalseExpr* equals **false**.

Control Constructs

Conditional Constructs

Logical Expressions Examples

```
123 > 99                          'Evaluates to true
"zero" < "one"                    'Evaluates to false
123 > 99 AND "zero" < "one"       'Evaluates to false
123 > 99 OR "zero" < "one"        'Evaluates to true
```

You can assign logical expressions to numeric variables, but most often they are evaluated within a control structure. True expressions = -1 and false expressions = 0.

Looping Constructs

See Also Example

A *loop* is a series of statements that repeat or loop. You can write a loop so that it repeats any number of times. Each repetition is called an *iteration*.

ScriptMaker contains three looping mechanisms:

- The For...Next loop repeats a sequence of statements a specified number of times. It is often used to search for characters in a string or elements in an array because the length of a string and the size of an array are easy to determine.
- The While...Wend and Do...Loop loops are controlled by a logical expression. They are called conditional loops because they repeat a sequence of statements until the logical expressions value changes. Use a conditional loop when the exact number of iterations is unknown. Often the exact number of records in a file is not known, so a conditional loop is used that stops when it determines that the end of the file has been reached.

When evaluating user input, the statements in a loop ask for user input, reject it when it is invalid, notify the user what is wrong, and repeat. A conditional loop allows the user to make as many attempts as necessary and terminates when valid data is input. However, if the input is a password or other sensitive data, for security reasons a **For** loop can limit the number of opportunities the user has to input the password incorrectly.

Control Constructs
Do...Loop
Exit For
Exit Do
For...Next
While...Wend

Looping Constructs Example

The following example shows a For loop nested inside a Do loop.

```
Dim Counter As Integer  ' For loop counter
Dim Max As Integer      ' For loop upper bound
Dim Answer As String    ' User input
Dim Total As Integer    ' Total of score
Dim Score As Integer    ' Input number
...
Do ' beginning of outer loop
   Total = 0
   Max = Val(InputBox$("How many scores in this group?"))

   For Counter = 1 To Max     ' beginning of inner loop
      Score = Val(InputBox$("Please enter a score:"))
      Total = Total + Score
   Next                       ' end of inner loop

   MsgBox  "The total for this group is " + Str$(Total)
   Answer = InputBox$("Do you have another group? (Y or N)")
Loop Until Answer = "N" or Answer = "n" ' end of outer loop
...
```

LTrim$( )

See Also Example

The LTrim$( ) function returns a string containing the specified string, but with leading spaces removed.

Syntax:

LTrim$(*exprS)*

exprS	A string expression from which to remove leading spaces.

Left$( )
Mid$( )
Right$( )
RTrim$( )
Trim$( )

LTrim$( ) Example

The following example demonstrates the use of LTrim$( ).

```
aString$ = "          10 leading spaces"

'Now remove the leading spaces
aString = LTrim$(aString)

'aString should now be equal to the string "10 leading spaces"
```

MailDocument

See Also Example

The MailDocument statement sends a specified file to network users. This statement is intended for use with Windows for Workgroups.

Syntax:

MailDocument *filename*

filename	A string expression containing the complete or relative pathname for the file to be mailed.

MailMsg
NetMsgAll( )
NetMsgSend

MailDocument Example

The following example mails D:\WORK\REVIEW\DRAFT2.DOC.

```
MailDocument 1, "d:\work\review\draft2.doc"
```

MailMsg

See Also Example

The MailMsg statement sends specified message text to network users. This statement is intended for use with Windows for Workgroups.

Syntax:

MailMsg *subject*, *message*

subject	A string expression containing the subject of the message.
message	A string expression containing the message to be sent

MailDocument
NetMsgAll( )
NetMsgSend

MailMsg Example

The following example sends a message with a subject of "MEETING CHANGE" and text of "Today's meeting will be held at 3 p.m. instead of 2 p.m."

```
MailMsg "MEETING CHANGE", "Today's meeting will be held at 3 p.m. instead of 2 p.m."
```

Main

See Also Example

Every script has a subroutine named Main. Main controls the scripts execution. It is the first to be executed and it causes other subroutines and functions to be executed by calling their names. Main calls the other subroutines and functions or they call each other. Main can be the only subroutine in the script, but when there are other subroutines and functions, it is the last one listed in the file.

Main starts with Sub Main. Mains last line ends the script: End Sub.

NOTE: If you are not writing long or complicated scripts, Main is probably the only subroutine in your script. Then the first line of your script is Sub Main.

Function...End Function
Sub...End Sub

Main Example

The following example shows a valid script with **Main** as the only subroutine.

```
Sub Main
   'Count to 100
   For i = 1 To 100
      ...
   Next i
End Sub
```

MCI( )

See Also Example

The MCI( ) function sends commands to the Media Control Interface (MCI), which is a high-level command interface to multimedia devices and resource files. Further information about MCI and MCI commands can be found in the *Multimedia Programmers Guide* and in the *Multimedia Programmers Reference* of the Windows 3.1 Software Development Kit (SDK).

The function returns the value 0 if the function was successful. Otherwise an error number is returned. When the function indicates an error.

Syntax:

MCI(*command*, *result*[, *error*])

command	A string expression containing the MCI command to be issued.
result	String variable that returns the value of the specified MCI command (if the command returns a value). Otherwise, it returns an empty string.
error	String variable that returns the text corresponding to the error (if one occurs). It returns an empty string when no error occurs.

Beep
PlayMedia
PlayMidi
PlaySound

MCI( ) Example

In the following example, the waveaudio device is opened with a waveform, played, and then closed. A message box displays the result of each command:

```
'Declare some variables
Dim result$, errorText$, returnValue%

'Open the waveaudio device with the chimes waveform
returnValue = MCI("open c:\windows\chimes.wav type waveaudio Alias
waveform", result, errorText)
MsgBox Str$(returnValue) + " " + result + " " + errorText

'Set the time format to samples
returnValue = MCI("set waveform time format samples", result, errorText)
MsgBox Str$(returnValue) + " " + result + " " + errorText

'Start playing from sample 1
returnValue = MCI("play waveform from 1", result, errorText)
MsgBox Str$(returnValue) + " " + result + " " + errorText

'Close the device
returnValue = MCI("close waveform", result, errorText)
MsgBox Str$(returnValue) + " " + result + " " + errorText
```

Menu

Overview See Also Example

The Menu statement selects a menu item from the active window. A run-time error occurs if the Menu statement specifies a menu item that does not exist or is not enabled. For example, "File" is the name of a pulldown menu and not a menu item. The menu item "File.foo" does not exist.

During a recording session with the Recorder, interactions with an applications menus generate Menu statements.

Syntax:

Menu *menuItem*

menuItem	A string expression containing the complete menu item name. See Menu Commands Overview for more information about menu item names.

Menu Overview
MenuItemChecked( )
MenuItemEnabled( )
MenuItemExists( )
WinActivate

Menu Example

The following examples show the use of the Menu statement.

```
'Select the Exit item from the File menu
Menu "File.Exit"

'Select Bold from the third level
Menu "Format.Character.Bold"

'Select the Maximize item from the Control menu
Menu ".Maximize"

'Select the second item from the File menu
Menu "File.#2"
```

Menu Overview

See Also

ScriptMaker provides you complete access to an applications pull-down menu system. All that you need to know is where the command you want to execute is located in the menu system of the application.

The ScriptMaker menu statements and functions are:

Menu *MenuItem*

MenuItemChecked(*MenuItem*)

MenuItemEnabled(*MenuItem*)

MenuItemExists(*MenuItem*)

All four of the menu statements apply to the active application and take the *MenuItem* parameter which is a string specifying the complete menu item name. Each successive menu level leading to the desired item is separated by a period. Menu item names are appended in order until the specified statement is reached. For example, the Open command on the File menu is represented by "File.Open". Cascading menu items have many periods, one for each menu, such as, "Format.Layout.Vertical". Menu items can also be specified using numeric index values. For example, to select the third item from the File menu, use "File.#3". To select the second item from the fourth menu use "#4.#2".

Items from the applications system menu can be selected by beginning the menu item name with a period. For example, ".Close" or ".Restore".

When processing menu item names, ScriptMaker ignores case, and removes all spaces, the ampersand "&", and all characters after a backspace or tab. For example, the menu item "&File.&Open" + Chr$(9)+"Ctrl+F12" translates to "File.Open". ASCII value 9 is the tab character.

NOTE: If the menu item includes an ellipsis (...) after the name, then do not include it when specifying it in the *MenuItem* parameter. For example, the Open menu item under the File menu is probably shown as "Open...", but if you specify "File.Open..." instead of "File.Open" for *MenuItem*, a run-time error occurs.

Dialog-box Controls Overview
Window Overview

MenuItemChecked( )

Overview See Also Example

Some menu items can be checked and unchecked to indicate one of two states. The MenuItemChecked( ) function returns TRUE if the menu item exists on the active applications menu system and is checked. Otherwise, the function returns FALSE.

Syntax:

MenuItemChecked(*menuItem*)

menuItem A string expression containing the complete menu item name. See Menu Overview for more information about menu item names.

Menu
MenuItemEnabled( )
MenuItemExists( )
WinActivate

MenuItemChecked( ) Example

In the following example, a message box finds out whether the "Edit.Word Wrap" menu item is checked in the active application.

```
If MenuItemChecked("Edit.Word Wrap") = TRUE Then
   MsgBox "Edit.Word Wrap is checked!"
Else
   MsgBox "Edit.Word Wrap is not checked!"
End If
```

MenuItemEnabled( )

Overview See Also Example

When a menu item is dimmed, the corresponding action is not enabled. The MenuItemEnabled( ) function returns TRUE if the menu item (or command) exists on the active applications menu system and is enabled. Otherwise, the function returns FALSE.

Syntax:

MenuItemEnabled(*menuItem*)

menuItem A string expression containing the complete menu item name. See Menu Overview for more information about menu item names.

Menu
MenuItemChecked( )
MenuItemExists( )
WinActivate

MenuItemEnabled( ) Example

In the following example, a message box checks whether the Save command in the File menu is enabled in the active application.

```
If MenuItemEnabled("File.Save") = TRUE Then
   MsgBox "File.Save exists and is enabled!"
Else
   MsgBox "File.Save is not enabled!"
End If
```

MenuItemExists( )

Overview See Also Example

Selection of a nonexistent menu item using the Menu statement causes a run-time error. The MenuItemExists( ) function determines if the particular menu item (or command) exists on the active applications menu system. The function returns TRUE if the menu item exists. Otherwise, the function returns FALSE.

Syntax:

MenuItemExists(*menuItem*)

menuItem	A string expression containing the complete menu item name. See Menu Overview for more information about menu item names.

Menu
MenuItemChecked( )
MenuItemEnabled( )
WinActivate

MenuItemExists( ) Example

In the following example, a message box displays the result of checking whether the New command in the File menu exists in the active application.

```
If MenuItemExists("File.New") = TRUE Then
   MsgBox "File.New exists!"
Else
   MsgBox "File.New does not exist!"
End If
```

Mid$

See Also Example

The Mid$ statement changes a specified string by replacing a substring in it with characters from another string.

Syntax:

Mid$(*originalStr*, *startPos* [,*length*]) **=** *newStr*

originalStr	A string expression into which the new string will be inserted. After the statement, *originalStr* contains *newStr* (or the portion of it that replaces characters in *originalStr*).
startPos	A numeric expression indicating the starting position of the substring in *originalStr* that will be replaced by characters from *newStr*. The first character of *originalStr* is at position 1.
length	A numeric expression giving the number of characters to replace. The default is from the starting position to the end of the string.
newStr	A string expression that is to replace part of *originalStr*.

NOTE: If you use the **Mid$** statement with a replacement string that is the same length or shorter than the substring it is replacing, all of the characters of the new string are included in the result. Otherwise, the extra characters of the new string are not included in the result.

Left$( )
LTrim$( )
Mid$( )
Right$( )
RTrim$( )
Trim$( )

Mid$ Example

The following example replaces the substring "dog" in "My dog has fleas." with the substring "cat" so that the string becomes "My cat has fleas." This example assumes that you want to keep copies of both the source and target strings, so it makes a copy of the source string before using the Mid$( ) function.

```
DogString$ = "My dog has fleas."
CatString$ = DogString 'copies the source
Mid$(CatString, 4) = "cat" 'changes the copy
```

The next example replaces the substring "dog" with the substring "elephant". Because "elephant" is longer than "dog", you cannot use the Mid$ statement (or you can only substitute "ele" for "dog").

```
DogString$ = "My dog has fleas."
Length% = Len(DogString)
LeftEnd% = InStr(DogString, "dog") - 1
RightStart% = Length - LeftEnd - Len("dog")
ElephantString$ = Left$(DogString, LeftEnd) + "elephant" + Right$
(DogString, RightStart)
```

Mid$( )

See Also Example

The Mid$( ) function returns a string containing the substring of the original string starting at the specified position and containing the specified number of characters.

Syntax:

Mid$(*originalStr*, *startPos* [,*length*])

originalStr	A string expression from which to retrieve the substring.
startPos	A numeric expression giving the starting position from which to retrieve the substring. The first character of *originalStr* is at position 1.
length	A numeric expression giving the number of characters to retrieve. The default is to end with the last character of the string.

Left$( )
LTrim$( )
Mid$
Right$( )
RTrim$( )
Trim$( )

Mid$( ) Example

Assuming that RecordString contains a last name starting at position 1 of length 25 characters and a first name starting at position 26 of length 15 characters, the following example retrieves the first and last names from the string:

```
RecordString$ = "Smith********************Jane***********"
LastName$ = Mid$(RecordString, 1, 25)
FirstName$ = Mid$(RecordString, 26, 15)

'At this point:
'  LastName = "Smith********************"
'and  FirstName = "Jane***********"
```

Minute( )

See Also Example

The Minute( ) function returns a number in the range from 0 to 59 representing the minute of the serial time.

Syntax:

Minute(*serialDateTime*)

serialDateTime	Serial time, a number of type double, from which the minute is to be extracted.

DateSerial( )
DateValue( )
Day( )
Hour( )
Month( )
Now( )
Second( )
TimeSerial( )
TimeValue( )
Weekday( )
Year( )

MkDir

See Also Example

The MkDir statement creates a directory. MkDir works in the same way as the DOS MD command. As in DOS, only one directory level can be created at a time.

Syntax:

MkDir *dir*

dir A string expression that contains a complete or relative path for the directory you want to create. A drive can also be specified, in which case the directory is created on the specified drive.

ChDir
ChDrive
Kill
Name...As
RmDir
FileCopy
FileMove

MkDir Example

The following example creates a directory named ASDF in the current directory:

```
MkDir "asdf"
```

The next example creates a directory on the C drive named ASDF:

```
MkDir "c:asdf"
```

MOD Operator

See Also Example

The MOD operator divides two whole numbers and results in the remainder. If an operand is in not a whole number, it is rounded before the division takes place.

Syntax:

operand1 **MOD** *operand2*

Operands:

operand1	A numeric expression in the range for longs for dividend.
operand2	A numeric expression in the range for longs for divisor.

* Operator
+ Operator
- Operator
/ Operator
\ Operator
^ Operator
Numeric Operator Precedence

MOD Operator Example

The following examples illustrate the behavior of the MOD operator.

The result is 1 because 3 divided by 2 leaves a remainder of 1.

```
z = 3 MOD 1.5
```

The result is 0 because 3 divided by 1 leaves a remainder of 0.

```
z = 3 MOD 1.4
```

Month( )

See Also Example

The Month( ) function returns a number in the range from 1 to 12 representing the month of the serial date.

Syntax:

Month(*serialDateTime*)

serialDateTime	Serial date, a number of type double, from which the month is to be extracted.

DateSerial( )
DateValue( )
Day( )
Hour( )
Minute( )
Now( )
Second( )
TimeSerial( )
TimeValue( )
Weekday( )
Year( )

Serial Date and Time Example

After calling the Now( ) function, you can extract the individual values from the date and time.

```
'Get the current date and time
serialDT# = Now( )

'Now extract the values
theMonth% = Month(serialDT)
theDay% = Day(serialDT)
theYear% = Year(serialDT)
theWeekday% = Weekday(serialDT)
theHour% = Hour(serialDT)
theMinute% = Minute(serialDT)
theSecond% = Second(serialDT)
```

MsgBox and MsgBox( )

Overview See Also Example

The MsgBox( ) function and statement allow you to display a predefined dialog box which contains:

- A message you specify for the user.
- One or more command buttons.
- An icon (such as a stop sign)
- A name for the dialog box.

The function returns the number of the command button selected by the user. The statement returns nothing.

Syntax for function:

MsgBox (*message* [, *type* [, *name*]] **)**

Syntax for statement:

MsgBox *message* [, *type* [, *name*]]

message	A string expression that the user must respond to.
type	A numeric expression that specifies some of the components and characteristics of the dialog box. It is the sum of the numbers (one from each of the groups shown below) that correspond to the command button combination, default button for the dialog box (the button executed if the user presses Enter), icon, and mode. The default value is 0, which specifies an OK button with no icons and that the script waits for the users response. When a number is out of range, the default is used.

Command button Combination	**Number**
OK (the default)	0
OK, Cancel	1
Abort, Retry, Ignore	2
Yes, No, Cancel	3
Yes, No	4
Retry, Cancel	5
Icon	**Number**
Stop	16
Question Mark	32
Exclamation Point	48
Information	64
Default Button	**Number**
First Button (the default)	0
Second Button	256
Third Button	512

Mode	Number
ScriptMaker application waits until user responds (the default)	0
All applications wait until user responds	4096

The default button is the button that is automatically selected if the user presses Enter.

name A string expression containing the title for the dialog box. The default title is BASIC.

The **MsgBox( )** function returns the number corresponding to the button selected by the user. The return values are as follows:

Command button	Return
OK	1
Cancel	2
Abort	3
Retry	4
Ignore	5
Yes	6
No	7

AnswerBox( )

MsgBox and MsgBox( ) Example

The following example displays the message Hello, world!.

```
MsgBox "Hello, world!"
```

When the following MsgBox( ) function is executed. Button_Choice will have the value 6 or 7, depending on whether the user selects the Yes or No command button.

```
Button_Choice = MsgBox("Get me off this ship!", 4+16, "THE TITANIC")
```

MsgClose

Overview The Progress Message Dialog Box See Also Example

The MsgClose statement closes the progress message dialog box. If the dialog box is not on-screen, no error occurs.

Syntax:

MsgClose

MsgOpen
MsgSetText
MsgSetThermometer

MsgOpen

Overview The Progress Message Dialog Box See Also Example

The MsgOpen statement specifies the location and components of the progress message dialog box and the maximum length of time it can be displayed. It also sends the dialog box its first progress message.

Syntax:

MsgOpen *message*, *timeout*, *isCancel*, *isThermometer* [, *x, y*]

message	A string expression to display as the message in the dialog box. The message can be changed with the MsgSetText statement.
timeout	Maximum number of seconds to display the dialog box. If set to 0, the dialog box is displayed until the user cancels, the MsgClose statement is executed, or the script ends.
isCancel	A numeric expression that determines whether a Cancel button appears. When TRUE, the Cancel button appears. When FALSE, no button appears.
isThermometer	A numeric expression that determines whether a a horizontal bar, called a thermometer, appears between the message text and the Cancel button. When TRUE, the thermometer appears. Initially it indicates 0% completion. It can be changed using the MsgSetThermometer statement.
x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in twips. The upper-left corner of the active window is 0, 0.

MsgClose
MsgSetText
MsgSetThermometer

MsgSetText

Overview The Progress Message Dialog Box See Also Example

The MsgSetText statement changes the message in the progress message dialog box. The dialog box is resized to accommodate the new text.

Syntax:

MsgSetText *message*

message	A string expression containing the message to be displayed in the progress message dialog box. A run-time error results if a dialog box is not currently on-screen.

MsgClose
MsgOpen
MsgSetThermometer

MsgSetThermometer

Overview The Progress Message Dialog Box See Also Example

The MsgSetThermometer statement changes the percentage displayed in the thermometer and the amount that the thermometer is filled in the progress message dialog box. It is ignored if the current progress message dialog box has no thermometer or if the dialog box is not currently on-screen.

Syntax:

MsgSetThermometer *percentage*

percentage	A number ranging from 0 to 100. The percent sign (%) is automatically inserted by ScriptMaker. A run-time error occurs if the percent value is outside of this range.

MsgClose
MsgOpen
MsgSetText

Name...As

See Also Example

The Name...As statement lets you rename a file. The functionality of this statement is identical to the functionality of the DOS REN command.

Syntax:

Name *oldFile* **As** *newFile*

oldFile	A string expression containing a complete or relative pathname of a file you want to rename.
newFile	A string expression containing a new name to give the file.

Kill
MkDir
RmDir
FileCopy
FileMove

Name...As Example

To rename the file TESTFILE as NEWFILE:

```
Name "testfile" As "newfile"
```

To rename the file ROOTFILE in the root directory of the C: drive as GOODFILE:

```
Name "c:\rootfile" As "goodfile"
```

NetAddCon

See Also Example

The NetAddCon statement connects a local disk drive or printer port to a network resource. A run-time error occurs if no network is present.

Syntax:

NetAddCon *netpath*, *password*, *localName*

netpath	A string expression that points to either a network print queue or else the server, volume, and path for a network directory.
password	A string expression containing the password required to access the *netpath* resource. If there is no password, this can be an empty string ("").
localName	A string expression identifying the local resource to be connected to the *netpath* resource. This expression can contain either a local printer port ("LPT1", "LPT2", or "LPT3") or a local drive (any drive from "A:" through "Z:").

NetAttach
NetBrowse$( )
NetCancelCon
NetDialog
NetGetCaps( )
NetGetCon$( )
NetGetUser$( )
NetLogin
NetLogout
NetShareAs
NetStopShare

NetAddCon Example

The following example maps the network directory that the user selects (in response to the NetBrowse$( ) function) to the local drive letter specified in response to the InputBox$( ) function.

```
'Get the user to select a network directory
net$ = NetBrowse$(0)
'Get the user to specify a local drive
prompt$ = "Enter the local drive letter for the selected network directory"
title$ = "Network Mapping"
local$ = InputBox$(prompt, title)
'Connect the local drive to the network path
NetAddCon net, "", local
```

NetAttach

See Also Example

The NetAttach statement creates an attachment between a computer, workstation, or other intelligent machine and a Novel NetWare file server. A run-time error occurs if no network is present.

Syntax:

NetAttach *server*

server	A string expression containing the name of the Novell file server to which an attachment is to be created.

NetAddCon
NetBrowse$( )
NetCancelCon
NetDetach
NetDialog
NetGetCaps( )
NetLogin
NetLogout
NetMapRoot
NetShareAs
NetStopShare

NetAttach Example

The following example attaches to a Novell Netware server named "mainserver."

```
NetAttach "mainserver"
```

NetBrowse$( )

See Also Example

The NetBrowse$( ) function displays a network dialog box and returns a pointer to the network resource the user selects with that dialog box. A run-time error occurs if no network is present.

Syntax:

NetBrowse$(*resourceType*)

resourceType	An integer identifying the type of network resources to be included in the dialog box that is displayed: 0 for network directories or 1 for network print queues.

NetAddCon
NetCancelCon
NetDialog
NetGetCaps( )
NetGetCon$( )
NetGetUser$( )
NetLogin
NetLogout
NetMapRoot
NetShareAs
NetStopShare

NetBrowse$( ) Example

The following example first uses the AnswerBox( ) function to prompt the user for the resource type, and then uses the selected resource type as the parameter for the NetBrowse function.

```
message$ = "Click Directory to select a network directory, or click Printer
to select a network print queue"
resource = AnswerBox (message, "Directory", "Printer", "Cancel")
If resource = 1 Then NetBrowse$(0)
If resource = 2 Then NetBrowse$(1)
```

NetCancelCon

See Also Example

The NetCancelCon statement breaks the connection to a network resource. A run-time error occurs if no network is present.

Syntax:

NetCancelCon *connection* [, *flag*]

connection	A string expression containing the name of either a network resource or the local drive or port connected to that resource. If *connection* is not found, an error occurs.
flag	A numeric expression: TRUE if *connection* is to be broken even if there are currently open jobs or files, or FALSE if *connection* is not to be broken if there are currently open jobs or files. The default *flag* is FALSE.

NetAddCon
NetAttach
NetBrowse$( )
NetDetach
NetDialog
NetGetCaps( )
NetGetCon$( )
NetGetUser$( )
NetLogin
NetLogout
NetShareAs
NetStopShare

NetCancelCon Example

The following example breaks the network connection for drive G:, unless files are currently open on that drive.

```
NetCancelCon ("G:")
```

The following example breaks the network connection for LPT1, unless jobs are currently being sent through that port.

```
NetCancelCon ("LPT1")
```

NetDetach

See Also Example

The NetDetach statement breaks the attachment between a computer, workstation, or other intelligent machine and a Novel NetWare file server. A run-time error occurs if no network is present.

Syntax:

NetDetach *server*

server A string expression containing the name of the Novell file server whose attachment is to be broken.

NetAttach
NetCancelCon
NetGetCon$( )
NetLogin
NetLogout
NetMapRoot
NetShareAs
NetStopShare

NetDetach Example

The following example detaches from a Novell Netware server named "mainserver."

```
NetDetach "mainserver"
```

NetDialog

See Also Example

The NetDialog statement displays the network driver's dialog box. The results of this statement depend upon the network driver. For example, this statement might allow a user to log onto the network or change some settings. A run-time error occurs if no network is present.

Syntax:

NetDialog

NetAddCon
NetBrowse$( )
NetCancelCon
NetDetach
NetGetCaps( )
NetGetCon$( )
NetGetUser$( )
NetLogin
NetLogout
NetMapRoot

NetDialog Example

The following example applies to Novell NetWare, for which this statement displays a dialog box for changing system settings. This example uses the AnswerBox( ) function to determine whether the user wants the network dialog box displayed.

```
message$ = "Click OK if you want to change your network driver settings"
response = AnswerBox (message)
If response = 1 Then NetDialog
```

NetGetCaps( )

See Also Example

The NetGetCaps( ) function returns an integer providing information about network capabilities; if no network is installed, this function returns 0.

Syntax:

NetGetCaps(*dataType*)

dataType An integer identifying the information desired:

- 1 Driver ID
- 2 Network type
- 3 Driver version
- 6 Connection capabilities
- 7 Print queue capabilities

The values that NetGetCaps( ) can return depend upon *dataType* and the network:

Data Type	Returned Value
1	Network driver specification
2	Type of network installed: 0 None 256 Microsoft Network 512 Microsoft LAN Manager 768 Novell NetWare 1024 Banyan Vines 1280 Tiara 10Net
3	Network driver version number
6	Bit mask; the sum of these capabilities: 1 Add a connection 2 Cancel a connection 4 Get a connection 8 Auto-connect via DOS 16 Browse dialog
7	Bit mask; the sum of these capabilities: 2 Open a print job 4 Close a print job 16 Hold a print job 32 Release a print job 64 Cancel a print job 128 Set the number of copies 256 Watch a print queue 512 Unwatch a print queue 1024 Lock print queue data 2048 Unlock print queue data 4096 Send queue-change messages to Print

Manager
8192 Abort a print job

NetAddCon
NetBrowse$( )
NetCancelCon
NetDialog
NetGetCon$( )
NetGetUser$( )
NetLogin
NetLogout

NetGetCaps( ) Example

The following example uses NetGetCaps(2) to determine what kind of network is installed, and then uses the MsgBox statement to display the information.

```
Dim netname$
net = NetGetCaps(2)
If net = 0 Then
   netname = "None"
ElseIf net = 256 Then
   netname = "MS Network"
ElseIf net = 512 Then
   netname = "MS LAN Manager"
ElseIf net = 768 Then
   netname = "Novell NetWare"
ElseIf net = 1024 Then
   netname = "Banyan Vines"
ElseIf net = 1280 Then
   netname = "Tiara 10Net"
Else
   netname = "Unknown"
End If
```

NetGetCon$( )

See Also Example

The NetGetCon$( ) function returns the name of the network resource currently connected to a local drive or device. A run-time error occurs if no network is present.

Syntax:

NetGetCon$(*localName*)

localName	A string expression identifying the network resource of interest. This expression can be either a local printer port ("LPT1", "LPT2", or "LPT3") or a local drive (any drive from "A:" through "Z:"). An error occurs if *localName* is not currently connected to a network resource.

NetAddCon
NetBrowse$( )
NetCancelCon
NetDialog
NetGetCaps( )
NetGetUser$( )
NetLogin
NetLogout
NetMapRoot
NetShareAs
NetStopShare

NetGetCon$( ) Example

The following example uses NetGetCon$( ) to determine what network resource LPT1 is currently mapped to, and then uses the MsgBox statement to display the mapping information.

```
local$ = "LPT1"
netpath$ = NetGetCon (local)
MsgBox "Local Name: " + local + "; Network Name: " + netpath
```

NetGetUser$( )

See Also Example

The NetGetUser$( ) function returns a string containing the login name of the user for the current network session. A run-time error occurs if a network is not present.

Syntax:

NetGetUser$ [( )]

NetAddCon
NetBrowse$( )
NetCancelCon
NetDialog
NetGetCaps( )
NetGetCon$( )
NetLogin
NetLogout
NetMemberGet( )
NetMemberSet
NetMsgAll
NetMsgSend

NetGetUser$( ) Example

The following example determines the login name of the current user, and uses MsgBox to display it.

```
userName$ = NetGetUser$( )
MsgBox userName
```

NetLogin

See Also Example

The NetLogin statement logs the specified user into the network. A run-time error occurs if no network is present or the wrong number of parameters is specified.

Syntax:

NetLogin *server*, *login*, *password*

server	A string expression containing the name of a network server or machine.
login	A string expression containing the login ID of the user to be logged in.
password	A string expression containing the login password for *login*.

AskPassword$( )
NetAddCon
NetAttach
NetBrowse$( )
NetCancelCon
NetDetach
NetDialog
NetGetCaps( )
NetLogout
NetShareAs
NetStopShare

NetLogin Example

The following example uses the AskBox$( ) function to get the user's login ID, the AskPassword$( ) function to get the user's password, and the NetLogin statement to log the user into a server named "mainserver". Note that this example avoids the display of the user's password, because AskPassword$( ) displays the user's input as asterisks.

```
login$ = AskBox$("Enter your login ID:")
password$ = AskPassword$("Enter your password:")
NetLogin "mainserver", login, password
```

NetLogout

See Also Example

The NetLogout statement logs out the current user from the specified network server. A run-time error occurs if no network is present or the wrong number of parameters is specified.

NOTE: Because logging out of a network server cancels all redirections of local resources to that server, NetLogout should be used with caution. A script that logs the user out of the network can cause problems, especially if the user is running Norton Desktop and/or Windows from a redirected drive.

Syntax:

NetLogout *server*

server	A string expression containing the name of the network server or machine from which the user is to be logged out.

NetCancelCon
NetGetCon$( )
NetGetUser$( )
NetDetach
NetLogin
NetShareAs
NetStopShare

NetLogout Example

The following example first uses the NetGetUser$( ) and AnswerBox( ) functions to determine whether the current user should be logged out, and then the NetLogout statement to perform the logout.

```
user$ = NetGetUser$()
message$ = "Click OK if you want to log out "
answer = AnswerBox(message + user)
If answer = 1 Then NetLogout "mainserver"
```

NetMapRoot

See Also Example

The NetMapRoot statement redirects a drive letter to be the root of the specified network path (Novell NetWare only). A run-time error occurs if a Novell NetWare network is not present or the wrong number of parameters is specified. (Before using this statement, use NetAddCon to connect to the desired network resource.)

Syntax:

NetMapRoot *drive*, *server\volume:path*

drive	A string expression containing the letter for a local drive (any drive from "A:" through "Z:").
server\volume:path	A string expression containing the Novell NetWare server name, volume name, and directory path to be mapped to *drive*.

NetAddCon
NetAttach
NetBrowse$( )
NetCancelCon
NetDetach
NetDialog
NetGetCaps( )
NetGetCon$( )

NetMapRoot Example

The following example uses the AskBox( ) function to prompt the user for the mapping information, and the NetMapRoot statement to perform the mapping.

```
local$ = AskBox$("Enter local drive letter")
net$ = AskBox$("Enter server\volume:path")
NetMapRoot(local, net)
```

NetMemberGet( )

See Also Example

The NetMemberGet( ) function returns TRUE if the current network user is a member of a specified group on a specified server, and otherwise returns FALSE. A run-time error occurs if a network is not present or if the user is connected to a network that does not support this function.

Syntax:

NetMemberGet(*server*, *group*)

server	A string expression containing the name of a network server or machine.
group	A string expression containing the name of the network group of interest.

NetGetUser$( )
NetLogin
NetMemberSet

NetMemberGet( ) Example

The following example uses the AskBox$( ) function to prompt the user for the name of a network group and then uses NetMemberGet( ) to see if the user is a member of that group on the server named "mainserver"; if not, the MsgBox statement displays a message that the current user (whose name is returned by the NetGetUser$( ) function) is not a member of the group.

```
group$ = AskBox$("Enter the name of the network group:")
If NetMemberGet("mainserver", group) = FALSE Then
   user$ = NetGetUser$( )
   MsgBox user + " is not a member of " + group
End If
```

NetMemberSet

See Also Example

The NetMemberSet statement assigns the current network user to the specified group on the specified server. A run-time error occurs if no network is present, if the user is connected to a network that does not support this statement, or if the user is not authorized to add users to groups.

Syntax:

NetMemberSet *server*, *group*

server	A string expression containing the name of a network server or machine.
group	A string expression containing the name of the network group to which the user is to be added.

NetGetUser$( )
NetLogin
NetMemberGet( )

NetMemberSet Example

The following example uses the NetGetUser$( ) and AskBox$( ) functions to prompt the user for the name of a network group, and then uses NetMemberSet to make the currently logged-in user a member of the specified group on a server named "mainserver".

```
user$ = NetGetUser$( )
message$ = "Add " + user + "to this network group:"
group$ = AskBox$(message)
NetMemberSet ("mainserver", group)
```

NetMsgAll

See Also Example

The NetMsgAll statement broadcasts a message to all users on the specified network server. A run-time error occurs if no network is present or the wrong number of parameters is specified.

Syntax:

NetMsgAll *server*, *message*

server	A string expression containing the name of a network server or machine.
message	A string expression containing the message to be sent to all users on *server*.

MailDocument
MailMsg
NetMsgSend

NetMsgAll Example

The following example sends logout warning messages to all users on a server named "mainserver".

```
message1$ = "Network maintenance begins in 15 minutes...Please log out!"
message2$ = "Network maintenance begins in 10 minutes...Please log out!"
message3$ = "Network maintenance begins in 5 minutes...Please log out!"
message4$ = "Network maintenance begins NOW...Please log out!"
NetMsgAll "mainserver", message1
Sleep 5000
NetMsgAll "mainserver", message2
Sleep 5000
NetMsgAll "mainserver", message3
Sleep 5000
NetMsgAll "mainserver", message4
```

NetMsgSend

See Also Example

The NetMsgSend statement sends a message to the specified user on the specified network server. A run-time error occurs if no network is present or the wrong number of parameters is specified. (What happens after this function is executed depends upon the network. For example, some networks will return an error message if the specified user is not logged in; other networks do not.)

Syntax:

NetMsgSend *server*, *user*, *message*

server	A string expression containing the name of a network server or machine.
user	A string expression containing the login ID of a user.
message	A string expression containing the message to be sent to *user*.

MailDocument
MailMsg
NetMsgAll

NetMsgSend Example

The following example sends a message to a user named "jdoe" on a server named "mainserver".

```
message$ = "Your home directory is getting very large. Please clean out old
files!"
NetMsgSend "mainserver", "jdoe", message
```

NetShareAs

See Also Example

The NetShareAs statement shares a specified local directory with other network users. This statement is intended for use with Windows for Workgroups.

Syntax:

NetShareAs *type*, *path*

type	The integer 1, indicating that a drive is to be shared.
path	A string expression containing the DOS directory path to be shared.

NetAddCon
NetAttach$( )
NetCancelCon
NetDetach
NetDialog
NetGetCon$( )
NetLogin
NetLogout
NetStopShare

NetShareAs Example

The following example shares D:\WORK\REVIEW with network users.

```
NetShareAs 1, "d:\work\review"
```

The following example shares the entire D: drive with network users.

```
NetShareAs 1, "d:"
```

NetStopShare

See Also Example

The NetStopShare statement discontinues sharing of a specified local directory with other network users. This statement is intended for use with Windows for Workgroups.

Syntax:

NetStopShare *type*, *path*

type	The integer 1, indicating a drive is to be unshared.
path	A string expression containing the DOS directory path to be unshared.

NetAddCon
NetAttach$( )
NetCancelCon
NetDetach
NetDialog
NetGetCon$( )
NetLogin
NetLogout
NetShareAs

NetStopShare Example

The following example unshares D:\WORK\REVIEW.

```
NetStopShare 1, "d:\work\review"
```

The following example unshares the entire D: drive.

```
NetStopShare 1, "d:"
```

NOT Operator

See Also Example

The NOT logical operator yields the logical negative of an expression. The result is TRUE if the relational expression and logical expression is FALSE. The result is FALSE if the expression is TRUE.

Syntax:

NOT *expr*

expr	A numeric, relational, or logical expression.

If the expression is numeric, the result is a bitwise NOT of the expression. If either of the expressions is a floating-point number, the two expressions are converted to longs before the bitwise NOT.

AND Operator
If...Then...Else...End If
OR Operator
XOR Operator

NOT Operator Example

The NOT operator can be used to test that a condition does not hold.

```
'Do not give free admission to anyone not named Darlene
If NOT (personName = "Darlene") Then
   freeAdmission = FALSE
End If
```

Now( )

See Also Example

The Now( ) function returns the current date and time in serial format, a number of type double.

Syntax:

Now( )

Date and Time Calculations
Date$( )
DateSerial( )
DateValue( )
Day( )
Hour( )
Minute( )
Month( )
Second( )
Time$( )
TimeSerial( )
TimeValue( )
Weekday( )
Year( )

Now( ) Example

The following example stores the current serial date and time in a variable.

```
serialDT# = Now( )
```

Null( )

See Also Example

The Null( ) function returns a null string that can be assigned to a string variable.

Syntax:

Null[( )]

Left$( )
LTrim$( )
Mid$( )
Right$( )
RTrim$( )
Trim$( )

Null( ) Example

The following example shows the use of Null to free the space that once stored a string.

```
'Assign a string to a string variable
guineaString$ = "We take up space!"

'Now make the string variable a null string
guineaString = Null
```

Oct$( )

See Also Example

The Oct$( ) function converts a specified decimal number to its octal, base 8, equivalent. The function returns a string, each character of which is a digit of the octal number.

Syntax:

Oct$(*exprN*)

exprN	A numeric expression (in the range for longs) indicating the decimal number to be converted to octal. The number is rounded to the nearest whole number before conversion.

Asc( )
Chr$( )
Hex$( )
Str$( )
Val( )

Oct$( ) Example

The following example converts the decimal number 16 to octal.

```
octOf16$ = Oct$(16)      'Result should be the string "20"
```

OKButton

Overview See Also Example

The OKButton statement defines an OK button that appears within a dialog box template. When the OK button is selected, the Dialog( ) function ends.

Syntax:

OKButton *x, y, width, height*

x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in dialog units. The upper-left corner of the window is 0, 0.
width, *height*	The integer expressions indicating the width and height of the dialog box in dialog units.

Begin Dialog...End Dialog
Dialog
Dialog( )

On Error

See Also Example

You use an On Error statement so that run-time errors do not stop the script. The statement specifies how run-time errors are to be handled. Both the On Error statement and the error-handling statements, if you are using them, must be in the subroutine or function where the error occurs.

Syntax:

On Error GoTo *label*

On Error Resume Next

On Error GoTo 0

label A valid identifier used to mark the statement to which control is transferred.

The script starts with the value of the most recent error set to 0 (which means no error has occurred so far). When an error occurs, the scripts error value changes to the number for that error. The scripts error value is reset to 0 by each Resume statement and as each function or subroutine ends.

If an error occurs within the error-handling statements, the error trapping is disabled and a run-time error terminates the script.

On Error GoTo *label*
On Error Resume Next
On Error GoTo 0
Control Constructs
Err
Err( )
Erl( )
Error
Error$( )
Resume

On Error GoTo

See Also Example

The On Error GoTo statement transfers control to the specified label when a run-time error occurs.

Syntax:

On Error GoTo *label*

label A valid identifier used to mark the statement to which control is transferred.

A label is an identifier followed by a colon (for example, ErrorHandler:). Following it is a sequence of statements that handles the error. The last error-handling statement is a Resume statement.

When using this type of error handling, the statement preceding the label should be either the Exit Function or Exit Sub statement. This keeps the routine from executing the sequence of error-handling statements as a regular part of the routine. The Resume statement should precede the End Sub statement or another *label*...Resume sequence.

Err( ), Error$( ), On Error GoTo, and Resume Example

The following example sends all errors to the same label. The statements between the label and the Resume statement display the error numbers and messages that ScriptMaker normally displays when a run-time error terminates a script. Err( ) is a predefined function that returns the value of the most recent error. Similarly, the Error$( ) function returns the error message associated with the most recent error.

```
Sub Main ( )
   On Error GoTo MessageDisplay
   ...
   cmd$ = "[CreateGroup(" + quoted(Setup.GroupName) + ")]"
   DDEExecute channel, cmd$
   ...
   Exit Sub
   'This routine can be used for all errors while you are
   'debugging
   'It may help you fix more than one error at a time
MessageDisplay:
   MsgBox Str$(Err( )) + Error$( )
   Resume Next
End Sub
```

The MsgBox statement in the example displays the error number 323 when the DDEExecute statement asks for a group to be created that already exists. Then the Error$( ) function returns the message "Process failed in other application." This can be a handy debugging tool. However, the message you get when a run-time error stops the scripts execution can have more information, such as the number of an array subscript that is causing a problem. In this example, the message you would have seen as the script came to a halt is: "Error 323 in line 42. Process failed in other application." ScriptMaker currently has no way to return the line number where the error occurs.

On Error Resume Next

See Also Example

An On Error Resume Next statement causes execution to continue with the line following the one that caused the error.

Syntax:

On Error Resume Next

When you are using this type of error handling, the statement *after* an error-prone statement should handle the error. If you are not exiting the subroutine or function as a result of the error, you should reset the error value to 0 by assigning 0 using the Err statement. Otherwise, an error you have already processed may be reprocessed by your next error-handling statement.

If you want to handle more than one possible error for a statement, you can use a Select Case statement. Each error number would be a different case.

NOTE: Use error-handling statements after every statement that can produce an error. Otherwise, the error handling for one statement may be processing an error that was caused by an earlier statement.

Control Constructs
Err
Err( )
Error
Error$( )
On Error
Resume

On Error Resume Next Example

In the following example, the script returns to the calling routine when an error is detected in the called routine.

```
Sub Group (ByVal channel%)
   On Error Resume Next
   ...
   cmd$ = "[CreateGroup(" + quoted(Setup.GroupName) + ")]"
   DDEExecute channel, cmd$
   If Err( ) <> 0 Then Exit Sub
   ...
End Sub
```

In the following error-handling statement, the error makes it necessary to exit a **For** loop. The error value is reset because the subroutine continues.

```
For
   ...
   If Err( ) <> 0 Then
      Exit For
      Err = 0
   End If
   ...
Next
```

On Error GoTo 0

See Also Example

An On Error GoTo 0 statement turns off the previously set method of error handling.

Syntax:

```
On Error GoTo 0
```

Control Constructs
Err
Err( )
Error
Error$( )
On Error
Resume

On Error GoTo 0 Example

In the following example, error-handling is turned off when the condition is true.

```
If condition
   On Error GoTo 0
   ...
End If
```

Open

See Also Example

Before you can read or write to a file, you must open it using the Open statement.

Syntax:

Open *filename* [**For** { **Input** | **Output** | **Append** }] **As** [#]*fileNum*

filename	A string expression containing the complete or relative pathname of the file you want to open.
fileNum	An integer expression, ranging from 0 to 255, that uniquely identifies the open file within your script. Subsequent operations on this file use this number to specify the file instead of the filename. Placing a # before the *fileNum* is optional, but may improve readability.

A file can be opened in one of three modes: input, output, or append.

- In input mode, you can only read information from the file.
- In output mode, you can only write information to the file. If the file you specify already exists, its previous contents are deleted. You start with a file of zero length any time you open a file in output mode.
- In append mode, as in output mode, you can only write information to the file. However, in append mode, if the file already exists, writing to the file adds data to the end of the files current contents. If no mode is specified, then the mode defaults to append.

CAUTION: When you open a file in output mode, you are not notified if the file already exists.

If a file is opened in input or output mode, the file pointer is positioned at the beginning of the file. If a file is opened in append mode, the file pointer is positioned at the end of the file.

Close
FreeFile( )
Input #
Input$( )
Line Input #
OpenFileName$( ) and SaveFileName$( )
Print #
Write #

Open Example

To open the file TESTFILE in append mode as file number 250:

```
'No mode is specified, so the default is append
Open "TESTFILE" As #250
```

To open the file TESTFILE in input mode as file number 0:

```
Open "TESTFILE" For Input As #0
```

OpenFileName$( ) and SaveFileName$( )

Overview See Also Example

The OpenFileName$( ) and SaveFileName$( ) functions give you access to two of Windows common dialog boxes. These dialog boxes provide an easy and familiar way to prompt the user for a filename when opening or saving a file. These functions cause the common file dialog boxes to appear, and return the complete DOS pathname for the file the user selects, or an empty string if the user cancels the dialog box. They do not open or save files. Your script must provide that functionality.

You supply the title for each dialog box and the types of files (along with the filters that locate them) that appear in the dialog box. Both dialog boxes are identical in functionality except that, if the user selects a file that already exists from the dialog box for saving files, the user is prompted whether or not to replace the file. The dialog boxes differ in appearance in that the list box for selecting the type of files to filter for is labeled List Files of Type (when opening files) and Save Files of Type (when saving files). The File Name list box is dimmed when you are saving files.

Syntax:

OpenFileName$(*name*, *extension*)

SaveFileName$(*name* [, *extension*])

name	A string expression that appears as the dialog boxs name.
extension	A string expression that specifies the available file types. The string should be in the following format: "*type*:*ext*[, *ext*][; *type*:*ext*[, *ext*]]..." where *ext* is a valid file filter such as *.BAT or *.?F?, Each *type:ext[, ext]* combination becomes a separate line in the List Files of Type drop-down list box. *Type* is a string identifying the type of files the filter locates, such as "Documents". *Ext* is any valid DOS extension, such as *.BAT or *.?F?

NOTE: The common file dialog boxes have a drop-down combination box that holds the file types specified as the *extension* parameter. Initially, when the dialog box appears, only those files having the extensions specified as the first type in the list of extensions is shown in the file list.

OpenFileName$( ) Example
SaveFileName$( ) Example

Open

OpenFileName$( ) Example

The following example uses the OpenFileName$( ) function to locate all your ScriptMaker scripts. All the files with the extension .SM appear in any directory selected using the Drives and Directories list boxes.

```
FileTypes$ = "All Scripts:*.SM"
SelectedFile$ = OpenFileName$("Open ScriptMaker Script", FileTypes)
If SelectedFile = "" Then
   MsgBox "No file was selected!"
Else
   MsgBox "The file " + SelectedFile + " was selected."
End If
```

Option Base

See Also Example

If you would prefer to use 1 as the automatic lower bound for subscripts in an array (instead of 0), use the Option Base statement. The Option Base statement must be used outside of a user-defined function or subroutine and is valid for all the subroutines and functions that follow it.

Syntax:

Option Base { 0 | 1 }

The parameter can be either 0, meaning to use a default lower bound of 0, or 1, meaning to use a default upper bound of 1.

ArrayDims
ArraySort
Dim
LBound( )
ReDim
UBound( )

Option Base Example

The following example uses Option Base to set 1 as the default lower bound for array declarations.

```
Option Base 1
Sub Main( )
  Dim MonthArray (12)    'contains elements 1 to 12
End Sub
```

OptionButton

Overview See Also Example

The OptionButton statement defines an option button with the specified text that appears within a dialog box template.

Syntax:

OptionButton *x, y, width, height*, *name*

x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in dialog units. The upper-left corner of the window is 0, 0.
width, *height*	The integer expressions indicating the width and height of the dialog box in dialog units.
name	String variable or literal for name of option button. It can contain an ampersand & in front of the character to be used as an accelerator key.

Begin Dialog...End Dialog
Dialog
Dialog( )

OptionButton and OptionGroup Example

The following script displays a dialog titled Flavors containing three option buttons and an OK command button.

```
Sub Main( )

   Dim flavors$(2)

   flavors(0) = "Chocolate"
   flavors(1) = "Vanilla"
   flavors(2) = "Strawberry"

   Begin Dialog OptionDialog 15,24,100,81, "Flavors"
      OptionGroup .Flavor
            OptionButton 5,5,90,14, flavors(0)
            OptionButton 5,25,90,14, flavors(1)
            OptionButton 5,45,90,14, flavors(2)
      OKButton 55,64,41,14
   End Dialog

   Dim FlavorDialog As OptionDialog

   Dialog FlavorDialog

   'What flavor option was selected
   MsgBox flavors(FlavorDialog.Flavor)

End Sub
```

OptionEnabled( )

Overview See Also Example

OptionEnabled( ) determines whether the option button with the specified name or ID is enabled in the active window or dialog box. This allows you to avoid the run-time error that occurs if a statement is executed for an option button that is disabled (dimmed). The function returns TRUE if the option button is enabled and FALSE if the option button is dimmed. If the option button does not exist in the current dialog box, a run-time error occurs.

Syntax:

OptionEnabled(*name* | *ID*)

name	A string expression containing the name of the option button.
ID	An integer that identifies the option button.

OptionExists( )
SetOption
GetOption( )
ButtonEnabled( )
CheckBoxEnabled( )
ComboBoxEnabled( )
EditEnabled( )
ListBoxEnabled( )

OptionExists( ), OptionEnabled( ), and SetOption Example

The following example checks if the option button named "1.44MB" both exists and is enabled before it selects it.

```
If OptionExists("1.44MB") = TRUE Then
   If OptionEnabled("1.44MB") = TRUE Then
      SetOption "1.44MB"
   End If
End If
```

In the following example, an option button is clicked:

```
WinActivate "Control Panel|Desktop"
'Click the "Center" option button
SetOption "Center"
```

OptionExists( )

Overview See Also Example

OptionExists( ) checks for the existence of an option button with the specified name or ID in the active window or dialog box. This allows you to avoid the run-time error that occurs if a statement is applied to an option button that does not exist. The function returns TRUE if the option button exists and FALSE otherwise.

Syntax:

OptionExists(*name* | *ID*)

name	A string expression containing the name of the option button.
ID	An integer that identifies the option button.

OptionEnabled( )
SetOption
GetOption( )
ButtonExists( )
CheckBoxExists( )
ComboBoxExists( )
EditExists( )
ListBoxExists( )

OptionGroup

Overview See Also Example

The OptionGroup statement signifies the start of a group of option buttons within a dialog box template. It also defines the name used to determine which option button (from the group of option buttons that follows) is selected when the Dialog( ) function ends.

Syntax:

OptionGroup *.field*

.field	An integer variable used to set and/or retrieve selected option button. Setting this field pre-selects one of the option buttons.

Begin Dialog...End Dialog
Dialog
Dialog( )

OR Operator

See Also Example

The OR logical operator yields the logical OR of two expressions. The result is TRUE if either or both relational or logical expressions are TRUE. If both expressions are FALSE, the result is FALSE.

Syntax:

expr1 **OR** *expr2*

expr1	A numeric, relational, or logical expression.
expr2	A numeric, relational, or logical expression.

If the expressions are numeric, the result is a bitwise OR of the two expressions. If either of the expressions is a floating-point number, the two expressions are converted to longs before the bitwise OR.

AND Operator
If...Then...Else...End If
NOT Operator
XOR Operator

OR Operator Example

The OR operator is usually used to see if at least one condition is satisfied.

```
'Give free admission to children, the elderly, and the handicapped
If age < 18 OR age > 65 OR handicapped = TRUE Then
   freeAdmission = TRUE
End If
```

Parameters

See Also

Parameters are values passed from one function or subroutine to another. The number and types of parameters in a function or subroutine call must match the number and types of those in the called routines declaration. They must also be in the same order. The called routine identifies each parameter by its order in the call and uses its own name for the parameter no matter what the name of the parameter is in the calling routine.

A parameter can be passed either by value or by reference.

Parameters are passed by reference unless explicitly passed by value. There are two ways to pass a parameter by value. One way involves the syntax of the call; the other involves the declaration of the called routine. See Parameters in Calls and Using Parameters in Function and Subroutine Declarations.

Parameters that are passed to the called routine because their values are needed by the called routine to complete its task are *input parameters*. Parameters whose values are determined by the called routine for the benefit of the calling routine are *output parameters*.

Input parameters are often passed by value to protect the original. Output parameters are always passed by reference. When a parameter is both an input and output parameter, it is passed by reference as well. Arrays, because of possible size and memory requirements, are always passed by reference.

Parameters in Calls

See Also Example

The syntax for parameters used in a call is different than the syntax for parameters used in a function or subroutine declaration.

Syntax:

A *parameter list* (see earlier syntax for functions and subroutines) is a series of zero or more parameters:

[*parameter* [**,** *parameter*]...]

The syntax for a parameter is:

[**(**] { *varName* | *expr* } [**)**]

You can force a parameter to be passed by value by putting parentheses around the parameter in the calling routine. This is in addition to the set of parentheses that may surround the entire parameter list. For example, Call Square(x) does not force x to be passed by value, but Call Square((x)) or its equivalent without the reserved word call Square(x) do force x to be passed by value.

When a parameter is passed by value, the parameter can be any expression. When it passed by reference, it should be the name of a variable. For example, the expression y + 7 can be passed by value, but not by reference, because it does not have a location in memory that can be accessed. The variable y can be passed either by value or by reference, because it is a variable name and, therefore, has a location in memory.

When a variable name is used as a parameter in the calling routine, it must already be declared (and therefore initialized) in the calling routine. Because it has already been declared, you never need use a type declarator or the **As** *Type* clause in a call.

To pass an entire array, you do not use empty parentheses after its name, as you would in the called routines declaration. To pass an element of an array, you use the subscripts that identify that element. Subscripts are always in parentheses. For example, if Array1 is an array, you can pass it to the Report subroutine with:

Call Report(Array1)

or

Report Array1

To pass an element of that array Array1 (2, 3) to the Square subroutine, you would use:

Call Square(Array1(2, 3))

or

Square Array(2, 3).

Parameters in Calls Examples

In the following assignment statement, the expression y + 7 is passed by value to a function.

```
x = Test((y + 7), z)
```

In the following subroutine call, z is passed by value.

```
Call Sort (x, y, (z))
```

PI

See Also Example

PI is a floating-point constant with a value of 3.14159265358979323846264338327 9. It is often used in trigonometric computations.

NOTE: **PI** can also be determined using the following formula:

```
4 * Atn(1)
```

Atn( )
Cos( )
Sin( )
Tan( )

PI Example

The following example determines the circumference of a circle given the radius.

```
Function Circumference(radius As Integer) As Double
   Circumference = 2 * PI * radius
End Function
```

PlayMedia

See Also Example

The PlayMedia statement controls multimedia devices.

Syntax:

PlayMedia *command* [, *returnString*]

command	The string expression containing the command to be passed to the multimedia command interpreter.
returnString	The string expression returned by the multimedia driver.

Valid *command* strings depend upon the multimedia devices and drivers that are installed. Windows 3.1 includes a waveform device driver to play and record waveforms (.WAV files), and a sequencer device driver to play and record MIDI files.

Many multimedia drivers accept the WAIT and NOTIFY parameters in *command*. These parameters have the following effects:

WAIT causes the system to stop processing input until the specified operation is completed. The user cannot switch to another task while the operation is being performed.

NOTIFY suspends script processing until the specified operation is completed. In the meantime, the user can perform other tasks and switch tasks.

WAIT NOTIFY performs the same way as WAIT.

MCI( )
PlayMidi
PlaySound
Beep

PlayMedia Example

The following example plays a compact disc on a CD audio player.

```
PlayMedia "open cdaudio shareable alias music notify"
PlayMedia"set music time format tmsf"
PlayMedia"play music from 1"
PlayMedia"close music"
```

PlayMidi

See Also Example

The PlayMidi statement plays .MID or .RMI sound files for devices that support the Musical Instrument Digital Interface. MIDI-compatible hardware as well as its device driver must be installed for this function to work.

Syntax:

PlayMidi *filename* [, *flag*]

filename	A string expression containing a complete or relative pathname for the .MID or .RMI file to be played. It cannot contain wildcards (? or *). If the pathname is not specified, the current directory, then the DOS path, are searched for the file. An error occurs if the file is not found.
flag	A numeric expression: TRUE if control is to return to the script as soon as the sound file begins playing, or FALSE if script processing is to be suspended until the file has finished playing. The default is FALSE.

MCI( )
PlayMedia
PlaySound
Beep

PlayMidi Example

The following example starts playing CANYON.MID and returns control to the script while the file continues to play.

```
PlayMidi "canyon.mid", TRUE
```

PlaySound

See Also Example

The PlaySound statement plays a .WAV sound file. This statement requires the presence of hardware that is compatible with Windows 3.1.

Syntax:

PlaySound *filename* [, *flag*]

filename A string expression containing either a complete or relative pathname for the .WAV file to be played, or a keyname from the [Sounds] section of the WIN.INI file. The string cannot contain wildcards (? or *).

If *filename* is a file and the pathname is not specified, the current directory, then the DOS path, are searched for the file; if the file is not found, the .WAV file assigned to the SystemDefault keyname, in the [Sounds] section of the WIN.INI file, is played.

If *filename* is a WIN.INI keyname, the .WAV file assigned to that keyname is played.

flag The integer for the bit, or the sum of the bits, representing the desired behavior:

- 0 Wait for the sound file to finish processing before continuing with script processing. This is the default.
- 1 Don't wait; process more statements as soon as the sound file starts.
- 2 If the specified sound file is not found, do not play a default.
- 8 Play the sound file repeatedly until PlaySound "", 0 is executed.
- 16 If another sound is already playing, ignore this request; do not interrupt the sound file that is currently playing.

A *filename* of "" (an empty string) and a *flag* of 0 cause a sound file to stop playing.

Because *flag* is a bit mask, you can add the bits together to get the desired results (except, of course, for bit 0). For example, a *flag* of 3 combines the effect of bit 1 with the effect of bit 2; a *flag* of 27 combines the effects of all the bits.

MCI( )
PlayMedia
PlayMidi
Beep

PlaySound Example

The following example plays CHIMES.WAV and suspends script processing until the file is finished playing.

```
PlaySound "chimes.wav"
```

PO_LANDSCAPE

See Also Example

PO_LANDSCAPE is a numeric constant with a value of 2.

The PrinterGetOrientation( ) function returns this value to indicate that the page orientation is landscape.

PO_LANDSCAPE is also used in the call to PrinterSetOrientation to set the page orientation to landscape.

PO_PORTRAIT
PrinterGetOrientation( )
PrinterSetOrientation

PO_PORTRAIT

See Also Example

PO_PORTRAIT is a numeric constant with a value of 1.

The PrinterGetOrientation( ) function returns this value to indicate that the page orientation is portrait.

PO_PORTRAIT is also used in the call to PrinterSetOrientation to set the page orientation to portrait.

PO_LANDSCAPE
PrinterGetOrientation( )
PrinterSetOrientation

PopupMenu( )

Overview See Also Example

The PopupMenu( ) function displays a list of choices as a pop-up menu which appears at the current location of the mouse cursor. It returns the subscript of the selected element of the array. If the user cancels by pressing Esc or Alt, the function returns a number that is one less than the lower bound for subscripts in the array.

Syntax:

PopupMenu(*menuItems*)

menuItems	The name of a one-dimensional array of strings, each element of which is a menu item. Wherever you want a separator bar on the menu, assign no value or an empty string ("") to the corresponding element.

Only one pop-up menu can be displayed at a time. A run-time error results if another script executes this function while a pop-up menu is visible.

SelectBox( )

PopupMenu( ) Example

The following use of PopupMenu( ) displays a list of applications. Array element 4 is empty, so a separator bar separates the utilities from the word processors.

```
Dim MyMenu$(1 To 6)
MyMenu(1) = "Norton Disk Doctor"
MyMenu(2) = "Norton Speed Disk"
MyMenu(3) = "Norton Diagnostics"
MyMenu(5) = "Microsoft Word"
MyMenu(6) = "WordPerfect"
Users_Choice = PopupMenu(MyMenu)
```

Predefined Dialog Boxes Overview

See Also

ScriptMakers predefined dialog boxes provide an easy means of providing information to the user or obtaining input from the user such as typed text, a password, or yes and no responses. The statements and functions for calling the predefined dialog boxes are listed here:

Category	Statements and Functions	Contents You Can Specify, Default Contents, and Returns
message only	MsgBox statement	**Contents You Can Specify:** Message or prompt. Combinations of standard command buttons, but best to use only OK. Icon. Title of dialog box. **Default Contents:** OK command button. BASIC as title of dialog box. **Returns:** Nothing.
message and command button response	MsgBox( ) function	**Contents You Can Specify:** Message or prompt. Combinations of standard command buttons. Icon. Title of dialog box. **Default Contents:** OK command button. BASIC as title of dialog box. **Returns:** Number corresponding to selected command button.
	AnswerBox( ) function	**Contents You Can Specify:** Message or prompt. Up to three user-defined command buttons. **Default Contents:** BASIC as title of dialog box. OK and Cancel command buttons. **Returns:** Number corresponding to selected command button.
text box and command button response	AskBox$( ) function	**Contents You Can Specify:** Message or prompt. Default contents of text box. **Default Contents:** Contents of text box. Empty string if user cancels.
	AskPassword$( ) function	**Contents You Can Specify:** Message or prompt. **Default Contents:** BASIC as title of dialog box.

		Displays text box. OK and Cancel command buttons. Displays asterisks for characters typed by user. **Returns:** Contents of text box. Empty string if user cancels.
	InputBox$ function	**Contents You Can Specify:** Message or prompt. Default contents of text box. Position of dialog box in current window. **Default Contents:** BASIC as title of dialog box. Displays text box. OK and Cancel command buttons. Has standard size. Takes only 12 lines of text. About 20 characters per line. **Returns:** Contents of text box. Empty string if user cancels.
selection from list	PopupMenu( ) function	**Contents You Can Specify:** Name and contents or array. **Default Contents:** Displays array elements as a pop-up menu that appears at the current location of the mouse cursor. **Returns:** Selection user makes from a pop-up menu.
	SelectBox( ) function	**Contents You Can Specify:** Title of dialog box. Name and contents or array for list box. One-line message for the user. **Default Contents:** BASIC as title of dialog box. Displays array elements in a list box inside of a dialog box that has OK, Cancel, and Help command buttons. **Returns:** Selection user makes from a list box.
modeless progress message	statements: MsgOpen MsgSetText MsgSet-Thermometer MsgClose	**Contents You Can Specify:** Title of dialog box. Name and contents or array for list box. One-line message for the user. **Default Contents:** BASIC as title of dialog box. Displays array elements in a list box

inside of a dialog box that has OK, Cancel, and Help command buttons.

Returns:

Can trap for error if user cancels.

All of ScriptMakers predefined dialog boxes, except for the progress message dialog box, are *modal*, which means that ScriptMaker stops executing statements until the user clicks one of the dialog boxs command buttons. With *modeless* dialog boxes, the script continues to execute statements while the dialog box is displayed.

Unless otherwise stated, each statement or function displays a dialog box which is sized to fit the message and the command buttons, but its maximum size is five-eighths of the width and three-fourths of the height of the screen. The widest button determines the width of the other buttons. When the message is long, it is word wrapped. In most dialog boxes, you can use Chr$(13) + Chr$(10) to include a carriage return/linefeed in the message when you want to specify more than one line. The font in the dialog box is eight-point Helvetica.

AnswerBox
AskBox$
AskPassword$
InputBox$
MsgBox
MsgBox
MsgClose
MsgOpen
MsgSetThermometer
MsgSetText
OpenFileName$)
PopupMenu
SaveFileName$
SelectBox
User-Defined Dialog Boxes Overview

Print

See Also Example

After opening a viewport window, text can be output to it using the **Print** statement.

Syntax:

Print *expr* [, | ;] [*expr* [, | ;]]...

expr Expressions to be printed.

Expressions to be printed are separated by either a comma (,) or a semicolon (;). The last expression can be followed by either a commam semicolon, or neither.

- A comma causes the next expression to print at the next *print zone*. A new print zone begins every 14 spaces.
- If you want to bypass some print zones, use an empty string as the expression for each print zone that you want to leave blank. For example, the following skips two print zones:

  print "", "",
- A semicolon causes the next expression to follow immediately after the current expression.
- If neither a comma nor a semicolon follows the last expression, a carriage-return/linefeed is written. The next expression prints at the beginning of the next line.

How data appears in the output:

- A string expressions are written without enclosing quotes.
- Numbers are written with a preceding space reserved for the sign. Negative numbers are preceded by a minus sign, while positive numbers are preceded with a space. A trailing space is added after the number for integers and longs. Singles are printed with 7 significant digits, while doubles are printed with 15 or 16 significant digits.

NOTE: If no viewport window is open at the time the Print statement executes, no output is generated. This condition may present itself undesirably when more than one script executing at the same time uses the viewport window. If the script that originally opened the viewport window terminates, the viewport window closes, causing further Print statements from other scripts executing at the same time to have no effect unless a new viewport window is opened.

Print #
ViewportClear
ViewportClose
ViewportOpen

Print

See Also Example

After a file has been opened in either output mode or append mode, the Print statement can be used to write information to the file. Each Print statement begins writing at the current location of the file pointer.

Syntax:

Print [# *fileNum*], *expr* [, | ;] [*expr* [, | ;]]...

fileNum — A numeric expression, from 0 to 255, that uniquely identifies a currently open file within your script.

Since the Print statement can also be used to write to a viewport window, the # character must precede the file number when the statement is used to print to files.

After the *fileNum*, the information to be printed is listed as a sequence of expressions. The following lists the features of the Print statement.

Expressions to be printed are separated by either a comma (,) or a semicolon (;). The last expression can be followed by either a comma or a semicolon, or neither.

- A comma moves the file pointer to the next *print zone*, so the first character of the next expression is written in the next print zone. A new print zone begins every 14 spaces.
- If you want to bypass some print zones, use an empty string as the expression for each print zone that you want to leave blank. For example, the following skips two print zones:

  Print #1, "", "",
- A semicolon does not move the file pointer, so the next expression should follow immediately after the current expression.
- If neither a comma nor a semicolon follows the last expression, a carriage-return/linefeed is written to the file. This positions the file pointer at the beginning of the next line.

How data appears in the file:

- A string expressions are written without enclosing quotes.
- Numbers are written with a preceding space reserved for the sign. Negative numbers are preceded by a minus sign, while positive numbers are preceded with a space. A trailing space is also added after the number for integers and longs. Singles are printed with 7 significant digits, while doubles are printed with 15 or 16 significant digits.

Input #
Input$( )
Line Input #
Open
Print
Seek
Write #

Print # Example

The following example prints the squares of the first ten positive numbers all on the same line of the open file:

```
Open "testfile" For Output As #1
For i = 1 To 10
   'The semicolon forces the next print to Print immediately after
   Print #1, i * i;
Next i
```

The file contains 1 4 9 16 25 36 49 64 81 100.

The next example prints the strings "asdf" and "qwer" to adjacent print zones and then issues a carriage-return/linefeed:

```
Open "testfile" For Output As #1
Print #1, "asdf", "qwer"
```

PrinterGetOrientation( )

See Also Example

The PrinterGetOrientation( ) function returns the current default page orientation, which is stored in the WIN.INI file in a section dedicated to the particular default printer. The function returns the constant PO_PORTRAIT if the current page orientation is portrait. It returns the numeric constant PO_LANDSCAPE if the current page orientation is landscape.

Syntax:

PrinterGetOrientation( )

PrintFile( )
PrinterGetOrientation( )

PrinterGetOrientation( ) Example

The following example determines whether the current page orientation is landscape or portrait.

```
If PrinterGetOrientation( ) = PO_LANDSCAPE Then
   MsgBox "Landscape"
Else
   MsgBox "Portrait"
End If
```

PrinterSetOrientation

See Also Example

The PrinterSetOrientation statement sets the page orientation to either portrait or landscape. The new mode is written to the WIN.INI file as the default page orientation.

Syntax:

PrinterSetOrientation *orientation*

orientation	A numeric expression containing the new page orientation for printing. It can be set to one of the following two constants:
	PO_PORTRAIT 1
	PO_LANDSCAPE 2

PrintFile( )
PrinterGetOrientation( )

PO_PORTRAIT, PO_LANDSCAPE, and PrinterSetOrientation Example

The following script displays an answer box for choosing the page orientation. After a selection is made, the script sets the page orientation accordingly.

```
Sub Main( )
   If AnswerBox("Orientation?", "Portrait", "Landscape") = 1 Then
      PrinterSetOrientation PO_PORTRAIT
   Else
      PrinterSetOrientation PO_LANDSCAPE
   End If
End Sub
```

PrintFile( )

See Also Example

The PrintFile( ) function sends the specified file to a Windows application to be printed. PrintFile( ) can send the file only if the following criteria are met:

- The script is executing in Windows version 3.1 or later.
- The file has an extension that has been associated with a Windows application.
- That application is registered with Windows for this purpose by its manufacturer. (The application appears in the Windows registration table.)

If an error occurs, the function returns a value less than 32. Otherwise, it returns the ID assigned to the print task and a message appears on the screen about the print task.

Syntax:

PrintFile(*filename*)

filename A string expression containing a complete or relative pathname for the file to be printed. It can contain wildcards (* and ?).

You cannot register additional applications, but you can associate additional extensions with applications that are already registered. An *association* is a relationship created between an extension and the application using the Association dialog box. It implies that all the files with the specified extension are of the type recognized by the application.

To see what applications are registered:

1. Choose ASSOCIATE... from the File Manager or Norton Desktop File menu.

The Associate dialog box appears.

2. Select various applications from the Associate With list box.

The words (Not registered) appears at the bottom of the dialog box for each application that is not registered. Otherwise, the name of the file type for which the application is registered appears. For example, you may see Microsoft Excel Chart, Calendar File, or Write Document.

3. Click OK.

To associate an extension with an application:

1. Choose ASSOCIATE... from the File Manager or Norton Desktop File menu.

The Associate dialog box appears.

2. Select an application from the Associate With list box.
3. Type an extension in the Extension text box.
4. Click OK.

The specified application, if registered, will be launched to print files ending with the additional extension. If the file is the type for which the application is registered, it will be printed.

Shell( )
PrinterGetOrientation( )
PrinterSetOrientation

PrintFile( ) Example

For example, when you install Windows 3.1, NOTEPAD.EXE is automatically registered to print a type of files called Text File and the .TXT extension is automatically associated with the application. When the PrintFile( ) function specifies a file with the extension .TXT, a search of the Windows registration table reveals that NOTEPAD.EXE can print the file, and NOTEPAD.EXE is launched with instructions for printing it.

```
'Print the file REPORT.TXT
taskID% = PrintFile("F:\REPORT.TXT")
```

The following example launches WINWORD.EXE (the Word for Windows executable file) to print FOODMENU.DOC. When you install Microsoft Word for Windows, it registers itself to print a type of files called Word Document and associates itself with the extension .DOCwhich makes this example successful.

```
'Print the file FOODMENU.DOC
result% = PrintFile("F:\FOODMENU.DOC")
```

The Progress Message Dialog Box

See Also Example

When the user doesnt see anything happening on the screen, he or she may think something has gone wrong with the script. You can let the user know that something really is happening by displaying the progress message dialog box. You can also show the percentage of completion by displaying a horizontal bar, called a thermometer, in the dialog box.

Unlike the other predefined dialog boxes, the progress message dialog box is modeless. That means the script can continue while the dialog box is being displayed. You can update the dialog box continually throughout the script.

Several statements control the progress message dialog box. MsgOpen and MsgClose make the dialog box appear and disappear. MsgSetText changes the message, and MsgSetThermometer changes the percentage of completion.

If you display the Cancel command button, the user can cancel the progress message dialog box. Canceling makes the progress message dialog box disappear and causes run-time error number 343, Message box canceled. This allows you to trap for that error if you have an On Error statement in your script and cancel the task being performed for the user.

Only one progress message dialog box can be on-screen at any one time. It disappears automatically when its script terminates.

Predefined Dialog Boxes Overview

Progress Message Dialog Box Example

This sample script uses the progress message dialog box to show the user the progress of a task. Where Sleep statements give the sample script a number of milliseconds to wait before executing the next statement, your script would have a series of tasks. Each message should explain what the task that follows it does. For example if the first task performed a search, the message in the MsgOpen statement might be "Searching...". If the last task printed a file, the message in the last MsgSetText statement might be "Printing...".

```
Sub Main
   MsgOpen "Hello", 20000, TRUE, TRUE, 1440, 2880
   Sleep 700
   MsgSetThermometer 25
   MsgSetText "Quarter Done"
   Sleep 700
   MsgSetThermometer 50
   MsgSetText "Half Done"
   Sleep 700
   MsgSetThermometer 75
   MsgSetText "Three-quarters Done"
   Sleep 700
   MsgSetThermometer 100
   MsgSetText "Good-bye"
   Sleep 700
   MsgClose
End Sub
```

PushButton

Overview See Also Example

This statement defines a command button within a dialog box template. When a command button is selected, the Dialog( ) function ends.

Syntax:

PushButton *x, y, width, height*, *name*

x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in dialog units. The upper-left corner of the window is 0, 0.
width, *height*	The integer expressions indicating the width and height of the dialog box in dialog units.
name	String variable or literal for name of command button. String can contain ampersand & in front of character to be used as accelerator key.

Begin Dialog...End Dialog
Dialog
Dialog( )

Dialog( ) and PushButton Example

The following script displays a dialog box containing eight buttons labeled with the compass directions and arranged in a circle. The Dialog( ) function returns the number of the selected button, which is then used as the subscript to display the text of the selected direction. The buttons are defined in clockwise order, which also determines their numbering, starting from the direction of North.

```
Sub Main( )

   Dim direction$(7)

   direction(0) = "N"
   direction(1) = "NE"
   direction(2) = "E"
   direction(3) = "SE"
   direction(4) = "S"
   direction(5) = "SW"
   direction(6) = "W"
   direction(7) = "NW"

   'Define 8 command buttons in a circle
   Begin Dialog DirectionsDialog 16,32,122,119, "Directions"
      PushButton 50,6,21,21, direction(0)
      PushButton 72,27,21,21, direction(1)
      PushButton 93,48,21,21, direction(2)
      PushButton 72,70,21,21, direction(3)
      PushButton 50,91,21,21, direction(4)
      PushButton 29,70,21,21, direction(5)
      PushButton 8,48,21,21, direction(6)
      PushButton 29,27,21,21, direction(7)
   End Dialog

   Dim DirDialog As DirectionsDialog

   'Which direction was selected?
   MsgBox direction(Dialog(DirDialog)-1)

End Sub
```

QueEmpty

See Also Example

The QueEmpty statement empties the event queue without first having to play the events contained in the queue.

Syntax:

QueEmpty

NOTE: Using a QueFlush statement immediately after a **QueEmpty** statement plays no events, because the queue is already empty.

QueFlush
QueKeys
QueMouseClick
QueMouseMove

QueEmpty Example

The following example empties the queue either with or without playing the events depending on the value of the logical expression Ready. If Ready is TRUE, the events are played by QueFlush. If Ready is FALSE, the events are not played.

```
If Ready Then
   QueFlush TRUE
Else
   QueEmpty
End If
```

QueFlush

See Also Example

The QueFlush statement plays the events in the event queue, which empties the queue. During a recording session with the Recorder, the recorder generates QueFlush statements whenever a statement, such as AppMove, occurs that cannot be placed in the event queue. QueFlush can play keystrokes and mouse events into any Windows application including a DOS application running in a window. The statement in the script that follows QueFlush is not executed until all the keystrokes and mouse events are performed.

Syntax:

QueFlush *saveStates*

saveStates	A numeric expression: either TRUE or FALSE. When TRUE, the states of CAPSLOCK, NUMLOCK, SCROLL LOCK, and INSERT prior to playing the events in the event queue are restored after **QueFlush** is complete. When FALSE, the states of these keys are left as they are after playing the events in the event queue.

QueEmpty
QueKeys
QueMouseClick
QueMouseMove

QueFlush Example

Play back events in the queue and save states:

```
QueFlush TRUE
```

QueKeyDn

See Also Example

QueKeyDn places a key down event into the event queue. The Recorder generates a QueKeyDn statement when key is pressed but not released immediately. Key down events are usually paired with a subsequent key up event (which releases the key).

Syntax:

QueKeyDn *keyStr*

keyStr	A string expression containing the name of the key that is pressed down. Keystroke Specification Format describes the format for specifying a keystroke.

DoKeys
QueFlush
QueKeys
QueKeyUp
SendKeys

QueKeyDn and QueKeyUp Example

In the following example, the Shift key is held down, the mouse dragged, then the Shift key is released:

```
'Hold down the Shift key
QueKeyDn "{+}"

'Press the left button and start dragging
QueMouseDn VK_LBUTTON, 204, 103
'Let go of the mouse
QueMouseUp VK_LBUTTON, 110, 103
'Let go of the Shift key
QueKeyUp "{+}"
QueFlush TRUE
WinActivate "Norton Desktop"
QueMouseUp VK_LBUTTON, 443, 350
QueFlush TRUE
```

QueKeys

See Also Example

QueKeys places complete keystrokes (when a keys is pressed and immediately released) into the event queue. The Recorder generates a QueKeys statement when keystrokes occur in conjunction with mouse events or partial keystrokes (such as QueMouseUp and QueKeyDn). It is more efficient to use QueKeys than DoKeys when other statement must go into the event queue.

Syntax:

QueKeys *keyStr*

keyStr	A string expression containing the keystrokes to put in the event queue. Keystroke Specification Format describes the format for specifying the keystrokes.

The QueFlush statement appears in the macro when a statement occurs that cannot go into the event queue such as WinActivate or AppMove or when the macro ends. When QueFlush is executed, the event queue is emptied.

NOTE: SendKeys also sends keystrokes to Windows applications, but it is not generated by the Recorder. It must be added manually to the script.

QueKeys Example

In the following example, the recorder records the NUMLOCK keystroke followed by the 1, 2, and then 3 keys typed on the numeric keypad, and finally the Enter key. The QueFlush statement then plays all five of the keystroke events:

```
QueKeys "{NUMLOCK}{NUMPAD1}{NUMPAD2}{NUMPAD3}{ENTER}"
QueFlush TRUE
```

In the next example, also generated by the recorder, the "a" key has been combined with the Ctrl key (indicated by the "^") , the Alt key (indicated by the "%"), and the Shift key (indicated by the capitalized "A"):

```
QueKeys "^(%(A))"
QueFlush TRUE
```

QueKeyUp

See Also Example

QueKeyUp places a key up event into the event queue. The Recorder generates a QueKeyUp statement when a key is released. Key up events are usually paired with a previous key down event.

Syntax:

QueKeyUp *keyStr*

keyStr The key that was released. Keystroke Specification Format describes the format for specifying the keystrokes.

DoKeys
QueFlush
QueKeyDn
QueKeys
SendKeys

QueMouseClick

See Also Example

QueMouseClick adds a single-click event, which consists of a mouse button pressed down and immediately released, to the event queue. The Recorder generates a QueMouseClick statement when a mouse single-click event occurs.

Syntax:

QueMouseClick *button*, *x*, *y*

button	A numeric constant indicating which mouse button generated the event. The left and right buttons are represented by the constants VK_LBUTTON and VK_RBUTTON, respectively.
x, *y*	Numeric expressions indicating the x- and y-coordinates (in pixels) for the location where the event occurred.

QueFlush
QueMouseDblClk
QueMouseDblDn
QueMouseDn
QueMouseMove
QueMouseUp
QueSetRelativeWindow

QueMouseClick Example

The following example shows the use of QueMouseClick.

```
'Left mouse button click at (x=167, y=205)
QueMouseClick VK_LBUTTON, 167, 205

'Play the click
QueFlush TRUE
```

QueMouseDblClk

See Also Example

QueMouseDblClk adds a double-click event, which consists of a single-click immediately followed by another single-click, to the event queue. The Recorder generates a QueMouseDblClk statement when a mouse double-click event occurs.

Syntax:

QueMouseDblClk *button*, *x*, *y*

button	A numeric constant indicating which mouse button generated the event. The left and right buttons are represented by the constants VK_LBUTTON and VK_RBUTTON, respectively.
x	Numeric expressions indicating the x- and y-coordinates (in pixels) for the location where the event occurred.

QueFlush
QueMouseClick
QueMouseDblDn
QueMouseDn
QueMouseMove
QueMouseUp
QueSetRelativeWindow

QueMouseDblDn

See Also Example

QueMouseDblDn adds a mouse double-down event, which consists of a mouse button pressed down and released followed by the mouse button pressed down again, to the event queue. The Recorder generates a QueMouseDblDn statement when a mouse button is clicked and then rapidly pressed back down.

Syntax:

QueMouseDblDn *button*, *x*, *y*

button	A numeric constant indicating which mouse button generated the event. The left and right buttons are represented by the constants VK_LBUTTON and VK_RBUTTON, respectively.
x, *y*	Numeric expressions indicating the x- and y-coordinates (in pixels) for the location where the event occurred.

QueFlush
QueMouseClick
QueMouseDblClk
QueMouseDn
QueMouseMove
QueMouseUp
QueSetRelativeWindow

QueMouseDblDn Example

The following example shows the use of QueMouseDblDn.

```
'Left mouse button double down (x=89, y=149)
QueMouseDblDn VK_LBUTTON, 89, 149

'Left mouse button up (x=100, y=149)
QueMouseUp VK_LBUTTON, 100, 149

'Play the double down and up
QueFlush TRUE
```

QueMouseDn

See Also Example

QueMouseDn adds a mouse button down event to the event queue. The Recorder generates a QueMouseDn statement when the mouse button is pressed down and held.

Syntax:

QueMouseDn *button*, *x*, *y*

button	A numeric constant indicating which mouse button generated the event. The left and right buttons are represented by the constants VK_LBUTTON and VK_RBUTTON, respectively.
x, *y*	Numeric expressions indicating the x- and y-coordinates (in pixels) for the location where the event occurred.

QueFlush
QueMouseClick
QueMouseDblClk
QueMouseDblDn
QueMouseMove
QueMouseUp
QueSetRelativeWindow

QueMouseDblClk, QueMouseDn, and QueMouseUp Example

The following example shows the Que statements recorded while editing in Word for Windows. The mouse actions were to double-click on a word and then drag it from one location to another in a document.

```
QueMouseDblClk VK_LBUTTON, 75, 260  'double-click left mouse button
QueMouseDn VK_LBUTTON, 75, 260      'press left mouse button
QueMouseUp VK_LBUTTON, 575, 344     'release left mouse button
```

QueMouseMove

See Also Example

The QueMouseMove statement adds a mouse movement to the event queue that indicates a new position for the mouse cursor. The Recorder generates QueMouseMove statements when the mouse pointer is moved.

Syntax:

QueMouseMove *x*, *y*

x, *y*	Numeric expressions indicating the x- and y-coordinates (in pixels) for the location where the event occurred.

QueFlush
QueMouseClick
QueMouseDblClk
QueMouseDblDn
QueMouseDn
QueMouseUp
QueSetRelativeWindow

QueMouseMove Example

The following example moves the mouse pointer to x = 100 and y = 100:

```
QueMouseMove 100, 100
```

QueMouseUp

See Also Example

QueMouseUp adds a mouse button up event to the event queue. The Recorder generates a QueMouseUp statement when a mouse button is released.

Syntax:

QueMouseUp *button*, *x*, *y*

button	A numeric constant indicating which mouse button generated the event. The left and right buttons are represented by the constants VK_LBUTTON and VK_RBUTTON, respectively.
x, *y*	Numeric expressions indicating the x- and y-coordinates (in pixels) for the location where the event occurred.

QueFlush
QueMouseClick
QueMouseDblClk
QueMouseDblDn
QueMouseDn
QueMouseMove
QueSetRelativeWindow

QueSetRelativeWindow

See Also Example

The QueSetRelativeWindow statement sets all subsequent mouse events relative to a specified window. The next QueFlush statement will use the new setting for playing mouse events stored in the event queue. After a QueFlush statement, mouse events are reset to be relative to the screen, unless another QueSetRelativeWindow is executed to give a new setting. The Recorder generates a QueSetRelativeWindow statement to specify the active window if the mouse relative to option was set to be the active window.

Syntax:

QueSetRelativeWindow *handle*

handle The handle to the window to which mouse events are to be relative. Using a handle with the value 0 makes the mouse events relative to the active window.

QueFlush
QueMouseClick
QueMouseDblClk
QueMouseDblDn
QueMouseDn
QueMouseMove
QueMouseUp

QueSetRelativeWindow Example

The following example adjusts mouse coordinates relative to Notepad.

```
'Get the handle to the Notepad window
hWnd = WinFind("Notepad")

QueSetRelativeWindow hWnd
```

Random( )

See Also Example

The Random( ) function returns a random number of type long that is greater than or equal to a specified minimum number and less than or equal to a specified maximum number.

Syntax:

Random(*min*, *max*)

min A numeric expression giving the minimum random number.

max A numeric expression giving the maximum random number.

Randomize
Rnd( )

Random( ) Example

The following call to Random( ) could be used to simulate the roll of a die.

```
'Generate a random number between 1 and 6
rollOfDie = Random(1,6)
```

Randomize

See Also Example

The Randomize statement initializes the random number generator with a new seed from which to generate random numbers. Repeating a seed value allows you to repeat a sequence of random numbers.

Syntax:

Randomize [*seed*]

seed	A numeric expression giving the new seed. If no seed is specified, then the current value of the system clock is used.

Random( )
Rnd( )

Randomize Example

In the following example, the seed for the random number generator is set to the current clock value and then a number is requested from 1 to 100.

```
Randomize
aRandomNumber = Random(1,100)
```

In the following example, the seed for the random number generator is set to 123 and then a number is requested from 1 to 100.

```
Randomize 123
aRandomNumber = Random(1,100)
```

ReadINI$( )

See Also Example

The ReadINI$( ) function returns a string containing the value of a particular entry in a specific section of the specified .INI file. It returns an empty string if the entry does not exist or does not have a value.

Syntax:

ReadINI$(*section*, *entry*[, *filename*])

section	A string expression containing the name of the section in the .INI file that contains the desired entry. Section names are specified without the enclosing brackets.
entry	A string expression containing the name of the entry whose value is to be retrieved.
filename	A string expression containing the complete or relative pathname for the .INI file to examine. The default file is the WIN.INI file. If no path precedes the name of the .INI file (for example, "CONTROL.INI"), it is assumed that the file is in the Windows directory. To examine a file not in the Windows directory, include a pathname (for example, ".\MYINI.INI" for the .INI file in the current directory, or "C:\TEST\TEST.INI" for the .INI file in the C:\TEST directory).

Environ$( )
ReadINISection
WriteINI

ReadINISection

See Also Example

The ReadINISection statement fills the specified array with all the entries in the specified section of the specified .INI file. Each element of the array contains a line of the .INI section. An empty array is returned if the section does not exist or does not contain any lines.

Syntax:

ReadINISection *section*, *entries*[, *filename*]

section A string expression that contains the name of the section in the .INI file that contains the desired entries. Section names are specified without the enclosing brackets.

entries The name of a one-dimensional string array to hold the entries found in the section.

This variable can be declared either as a dynamic array, such as Dim a$( ), or as an array with one dimension such as Dim a$(1 To 100). Any other type of string variable causes an error.

The statement redimensions the array to hold all of the directory names that match the given specification.

filename A string expression that contains the complete or relative pathname for the .INI file to read. The default file is the WIN.INI file.

If no path precedes the name of the .INI file (for example, "CONTROL.INI"), it is assumed that the file is in the Windows directory. To examine a file not in the Windows directory, include a pathname (for example, ".\MYINI.INI" for the .INI file in the current directory, or "C:\TEST\TEST.INI" for the .INI file in the C:\TEST directory).

You use the ArrayDims( ) function with *entries* to determine if the ReadINISection statement found any entries. The ArrayDims( ) function returns 0 if *entries* is empty. If the statement finds entries that match the specification, ArrayDims( ) returns the value 1. To find the lowest and highest subscripts for the elements in *entries*, and thereby the number of entries found, use the LBound( ) and UBound( ) functions. Even if you declare the array with a specified lower bound, that lower bound is not guaranteed to remain the lower bound if the array has been redimensioned.

NOTE: If ArrayDims is 0, using the **LBound( )** or **UBound( )** functions will cause errors because *entries* has no elements.

Environ$( )
ReadINISection
WriteINI

Recursion

See Also Example

When a function or subroutine calls itself, it is *recursive*.

Direct recursion takes place when a routine calls itself. *Indirect recursion* takes place when one routine starts a series of calls that result in some routine recalling the first routine. A simple example is one routine calling a second which in turn calls the first. When you use recursion, you must be sure that at some point the recursion stops. Like an infinite loop, recursion can occur over and over until no more memory is available.

User-Defined Functions and Subroutines

Recursion Example

The following example of a recursive subroutine calls itself to subtract 1 from a number for each recursive call it makes. The recursion stops when the number, a parameter to the call, is no longer positive. As the script returns from each call, the subroutine starts adding one to the number. Just before each recursive call, and right after the return from a recursive call, a message box displays the number. When executed, this script should help you visualize how recursion operates.

```
Sub CountDownAndUp(n As Integer)

   'Should we continue
   If n > 0 Then

      'Counting down
      MsgBox Str$(n)

      'Recurse to count down
      CountDownAndUp n - 1

      'Counting back up
      MsgBox Str$(n)

   Else

      MsgBox "0"

   End If

End Sub
```

ReDim

See Also Example

Sometimes the number of dimensions needed in an array is not known until runtime. Assigning the space for each dimension at runtime rather than at compile time is called *dynamic dimensioning*. A ReDim statement, which can occur only inside a subroutine or a function, is an executable statement that changes an array at runtime.

Syntax:

ReDim *arrayName* **(** [*subscripts*] **)** [**As** *type*] [**,** *arrayName* **(** [*subscripts*] **)** [**As** *type*]]...

arrayName	The name of the variable.
type	Integer, Long, Single, Double, or String. If you use a type declarator at the end of the variables name, the [**As** *type*] clause is unnecessary.
subscripts	The number of dimensions and the range of subscripts available in each dimension.

subscripts is defined as:

[*lowerBound* **To**] *upperBound* [**,** [*lowerBound* **To**] *upperBound*]...

The number of ranges provided indicates the number of dimension.

lowerBound	A numeric expression indicating the lowest subscript in a dimension.
upperBound	A numeric expression indicating the highest subscript in a dimension.

When the ReDim statement does not specifically set the lower bound for the subscripts in any dimension of an array, that lower bound is assumed to be 0 or the value set using Option Base.

The type of the arrays elements cannot be changed with the ReDim statement.

You can reuse an array by reinitializing its values to 0 (for arrays with numeric elements) and to an empty string (for arrays with string elements). The ReDim statement always initializes or reinitializes an array. Whatever was stored in the array previously is lost.

ArrayDims
ArraySort
Dim
LBound( )
Option Base
UBound( )

ReDim Example

Suppose the Main subroutine can call either of two subroutines to use an array. One subroutine may need three dimensions in the array and the other may need only two. You can use a **ReDim** statement, in the called subroutine, to specify the actual size no matter what dimensions were originally given to the array. For example:

```
ReDim DayArray(0 To 8,6 To 10)
```

Rem

See Also Example

The reserved word Rem comments a whole line. The line to be commented begins with the reserved word Rem.

Syntax:

Rem *comment*

comment

Rem Example

This line shows how to use **Rem**:

```
REM This script performs...
```

Reset

See Also Example

The Reset statement closes all open files.

Syntax:

Reset

Close
Open

Reset Example

In the following example, the single Reset statement closes both open files:

```
Open "testfil1" As #1
Open "testfil2" As #2
Reset
```

Resume

See Also Example

The Resume statement ends an error handling routine and continues execution.

Syntax:

Resume {[**0**] | **Next** | *label* }

label A valid identifier used to mark the statement to which control is transferred.

The Resume statement can transfer control to:

- The statement that caused the error (if you use Resume 0).
- The statement after the one that caused the error (if you use Resume Next).
- Another label (if you use Resume *label*).

Err
Err( )
Error
Error$( )
On Error

Right$( )

See Also Example

The Right$( ) function returns a string containing the specified number of ending or rightmost characters from the specified string. If *n* is greater than or equal to the number of characters in *exprS*, it returns the entire string.

Syntax:

Right$(exprS, n)

exprS	A string expression from which to retrieve characters.
n	The number of rightmost characters to retrieve from *exprS*.

Left$( )
LTrim$( )
Mid$( )
RTrim$( )
Trim$( )

Right$( ) Example

In the following example, assume that a percent sign (%) separates the first name from the last name in the string Name.

```
'Find the percent sign
Position% = InStr(Name, "%")
'Retain only the rightmost characters
LastName = Right$(Name, Position + 1)
```

RmDir

See Also Example

The RmDir statement deletes a directory from disk. It works much the same way as the DOS RD command. As in DOS, you can delete only an empty directory. You can use the Kill statement to delete files.

Syntax:

RmDir *dir*

dir A string expression that contains the complete or relative pathname for the directory. You cannot use wildcards (* and ?).

ChDir
ChDrive
Kill
MkDir
Name...As
FileCopy
FileMove

RmDir Example

The following example removes the directory named ASDF from the current drive:

```
RmDir "asdf"
```

The next example removes the directory named ASDF from the C drive:

```
RmDir "c:asdf"
```

Rnd( )

See Also Example

The Rnd( ) function returns a random number (of type single) between 0 and 1. If the expression is negative, the function always returns the same number. If zero, then the last number generated is returned. Otherwise, if greater than zero or omitted, generates the next random number.

Syntax:

Rnd[(*exprN*)]

exprN	A numeric expression. The default is a number greater than 0.

Random( )
Randomize

Rnd( ) Example

The following example uses the Rnd( ) function to randomly pick a whole number from a specified range.

```
'Range is from 0 to n
randomNumber% = Rnd( ) * n
```

RTrim$( )

See Also Example

The RTrim$( ) function returns a string containing the specified string, but with the trailing spaces removed.

Syntax:

RTrim$(*exprS)*

exprS A string expression from which to remove trailing spaces.

Left$( )
LTrim$( )
Trim$( )
Mid$( )
Right$( )

RTrim$( ) Example

The following example demonstrates the use of **RTrim$( )**.

```
aString$ = "10 trailing spaces          "

'Now remove the leading spaces
aString = RTrim$(aString)

'aString should now be equal to the string "10 trailing spaces"
```

SaveFileName$( ) Example

The following example uses the SaveFileName$( ) function to locate pictures.

```
FileTypes$ = "All Files:*.*;Bitmaps:*.BMP;Metafiles:*.WMF"
SelectedFile$ = SaveFileName$("Save Picture", FileTypes)
If SelectedFile = "" Then
   MsgBox "No file was selected!"
Else
   MsgBox "The file " + SelectedFile + " was selected."
End If
```

Initially, all files with extensions of *.BMP and *.WMF in the current directory are displayed in the file list of the dialog box. The user can manually type in a filename or select one from the file list. After the user exits the dialog box, the function returns a string value. If the user clicked Cancel, SelectedFile contains an empty string and a message box displays "No file was selected!" Otherwise, SelectedFile contains the complete pathname for the selected file and a message box displays that name.

Syntax

Example

The syntax used in ScriptMaker documentation is a variation of Backus-Naur Form (BNF), a standard method of indicating how a statement can be written correctly in a programming language.

Bold	Words in bold are reserved words that must be used in your statements and functions exactly as they appear in the syntax.
Italics	Words that appear in italics are parameters, and so forth. In your statements and functions, you substitute variable names, literals, and so forth (of the correct data type) for the words in italics. Below the syntax is an explanation of the words that appear in italics.
[...]	The brackets do not appear in your statements and functions. In the syntax, it means that anything between the brackets can be omitted and the statement will still compile. Usually, when you omit something, its default is used.
|	The vertical bar does not appear in your statements and functions. In the syntax, it means that you have a choice. No more than one of the choices will appear in your statement. When the choices are in brackets, making a choice is optional. When choices are not in brackets, you must pick one of them. Curly braces { ... } appear around the choices when you must make a choice and it is not syntactically clear what the choices are.
...	The ellipsis does not appear in you statements and functions. In the syntax, it means that the previous syntactical item can be repeated as many times as you like.
,	Commas that appear in the syntax are required in the statements and functions. Normally they separate parameters.
( )	Parentheses that appear in the syntax are required in the statements and functions. Normally they enclose parameters or the subscripts of an array.

Syntax Example

The following syntax (one of three possible syntaxes) for the Do...Loop control construct shows that:

- The reserved words Do and Loop must appear in the loop you write.
- The reserved word While or Until must appear in the loop you write. The curly braces make it clear that your choice is between While and Until--not Loop While and Until.
- Using executable statements between the reserved words Do and Loop is optional.
- If you use statements, any number of them can appear between the reserved words Do and Loop.
- You must write a logical expression to substitute for exprL.

Do
[*statement*]...
Loop { **While** | **Until** }*exprL*

exprL	A relational or logical expression.
statement	An executable statement.

The following syntax for the Line$( ) function shows that:

- The reserved word Line$ must appear in the function you write.
- Parentheses and commas that must appear in the function you write.
- The *text* parameter for which you substitute a string expression.
- The *first* parameter for which you substitute a numeric expression.
- The *last* parameter is optional. If you use it, you insert a numeric expression. If you don't use it, its default (1) is used by ScriptMaker..

Line$(*text*, *first*[, *last*])

text	A string expression containing the text to parse.
first	A numeric expression specifying the first line to retrieve. Line 1 is the first line of the text.
last	A numeric expression specifying the last line to retrieve. The default is 1 so one line is returned.

ScriptMakerHomeDir$( )

See Also Example

The ScriptMakerHomeDir$( ) function returns a string containing the name of the directory that ScriptMaker uses to locate files that are part of the ScriptMaker system.

Syntax:

ScriptMakerHomeDir$( )

ScriptMakerOS( )
ScriptMakerVersion$( )

ScriptMakerHomeDir$( ) Example

The following example stores ScriptMakers home directory in the string variable homeDir.

```
homeDir$ = ScriptMakerHomeDir$( )
```

ScriptMakerOS( )

See Also Example

The ScriptMakerOS( ) function returns a number that specifies the host operating environment:

0 indicates that Windows is the host operating system.

1 indicates that DOS is the host operating system.

Syntax:

ScriptMakerOS( )

ScriptMakerHomeDir$( )
ScriptMakerVersion$( )

ScriptMakerOS( ) Example

The following example stores a 0 in the variable opSys for the Windows operating system.

```
opSys% = ScriptMakerOS( )
```

ScriptMakerVersion$( )

See Also Example

The ScriptMakerVersion$( ) function returns a string containing the version of ScriptMaker as a major and minor version number (for example, "1.1").

Syntax:

ScriptMakerVersion$( )

ScriptMakerHomeDir$( )
ScriptMakerOS( )

ScriptMakerVersion$( ) Example

The following example stores ScriptMakers version number in the variable version.

```
version$ = ScriptMakerVersion$( )
```

Second( )

See Also Example

The Second( ) function returns a number in the range from 0 to 59 representing the second from a serial time.

Syntax:

Second(*serialDateTime*)

serialDateTime	Serial time, a number of type double, from which the second is to be extracted.

DateSerial( )
DateValue( )
Day( )
Hour( )
Minute( )
Month( )
Now( )
TimeSerial( )
TimeValue( )
Weekday( )
Year( )

Seek

See Also Example

The Seek statement moves the file pointer. This is useful when you want to read from or write to only a specific portion of a file. You position the file pointer at the start of the desired portion and then start reading or writing.

All read and write operations begin their input or output at the character position pointed to by the file pointer.

Syntax:

Seek [#] *fileNum*, *position*

fileNum	A numeric expression, from 0 to 255, that uniquely identifies the open file within your script.
position	A number in the range from 1 to 2,147,483,647 that gives the new position of the file pointer. Position 1 is the position of the first character of a file.

EOF( )
FileAttr( )
Input #
Input$( )
Line Input #
Loc( )
LOF( )
Open
Print #
Seek( )
Write #

Loc( ), Seek, and Seek( ) Example

Assume that the file TESTFILE contains nine lines of information, where each line has a two digit number, and a three character string. A comma separates the number and the string, and double quotation marks enclose the string.

```
12,"ABC"
23,"BCD"
34,"CDE"
45,"DEF"
56,"EFG"
67,"FGH"
78,"GHI"
89,"HIJ"
90,"IJK"
```

The number of characters on each line is 10. The two digit number uses two characters. The comma is one character. The string of three letters and the two enclosing double quotation marks come to five characters. And finally, the carriage-return/linefeed takes two additional characters. This makes 10 characters per line.

The first character of the file, which is also the first character of the first line, is at position 1, so the first character of the second line must be at position 11. The first character of the third line is at position 21, and so on.

The following script uses the Seek statement to go to the third line, which is at position 21, of the above file. Two lines are then read. Finally the Seek( ) function is used to determine the new position of the file pointer. The Loc( ) function could also have been used instead of the Seek( ) function.

```
Open "testfile" For Input As #1

'Seek to the third line (position 21)
Seek #1, 21

'Read in the line with 34,"CDE"
Input #1, num34%, strCDE$

'Read in the line with 45,"Def"
Input #1, num45%, strDEF$

'Determine the new position of the file pointer
curPos& = Seek(1) 'equivalent to curPos& = Loc(1)
Close #1
```

Seek( )

See Also Example

The Seek( ) function returns a number in the range from 0 to 2,147,483,647, indicating the current position of the file pointer.

Syntax:

Seek(*fileNum*)

fileNum	A numeric expression, from 0 to 255, that uniquely identifies a currently open file within your script.

EOF( )
FileAttr( )
Loc( )
LOF( )
Open
Seek

Select Case...End Select

See Also Example

A Select Case statement begins with the reserved words Select Case followed by a numeric or string expression and ends with the reserved words End Select. Between the Select Case and End Select statements are a series of Case statements, each of which is associated with a sequence of statements.

When the expression specified after the words Select Case matches one of the expressions specified after the word Case, the statements associated with that case are executed.

If no match is found, the statements associated with Case Else are executed. You can have only one Case Else. It always precedes the End Select statement. If there is no Case Else when no cases match, no sequence of statements is executed.

Syntax:

Select Case *testExpr*
[**Case** *caseExpr* [, *caseExpr*]...
[*statements*]...]...
[**Case Else**
[*statements*]...]
End Select

Syntax for *testExpr*, the test expression:

exprN | *exprS*	A numeric or string expression.

Syntax for *caseExpr*, the case expression:

exprN | *exprS*	A numeric or string expression, such as 2, **Val** (UserInput), or "red".
exprN **To** *exprS* | *exprN* **To** *exprS*	A range from one numeric or string expression to another of the same type, such as 1 **To** 10 or "a" **To** "z".
Is *RelOp* {*exprN* | *exprS*}	An open-ended range using a relational operator, such as **Is** > 40 or **Is** <= "zebra".
statements	Executable statements.

Conditional Constructs
If...Then...Else...End If

Select Case...End Select Examples

In the following example, a Select Case statement decides which sequence of statements to execute based on the value of a string.

```
...
Select Case Grade
   Case "A"
      ...   'sequence of statements
   Case "B" To "D"
      ...   'sequence of statements
   Case Is > "D"
      ...   'sequence of statements
   Case Else
      ...   'sequence of statements
End Select
```

The following example uses **Select Case** statements to determine what number or letter a user input. It is also an example of nesting **Select Case** statements in **If** statements.

```
...
If Val(UserInput$) = 0 Then    ' Is it a letter or number?
   Select Case Asc(UserInput$)' If it's a letter.
      Case 65 To 90      ' Must be uppercase.
            Msg$ = "You entered the uppercase letter '"
            Msg = Msg + Chr$(Asc(UserInput$)) + "'."
      Case 97 To 122     ' Must be lowercase.
            Msg = "You entered the lowercase letter '"
            Msg = Msg + Chr$(Asc(UserInput$)) + "'."
      Case Is = 61       ' Must be something else.
            Msg = "You entered an '=' sign"
   Case Else' Must be something else.
            Msg = "You did not enter a letter or a number."
   End Select
Else
   Select Case Val(UserInput$)' If it's a number.
      Case 1, 3, 5, 7, 9      ' It's odd.
            Msg = UserInput$ + " is an odd number."
      Case 0, 2, 4, 6, 8      ' It's even.
            Msg = UserInput$ + " is an even number."
      Case Else    ' Out of range.
            Msg = "You entered a number outside "
            Msg = Msg + "the requested range."
   End Select
End If
```

SelectBox( )

Overview See Also Example

The SelectBox( ) function allows you to display a predefined dialog box that contains:

- A list box.
- A one-line message.
- The name of the dialog box.
- The OK and Cancel command buttons.

The function returns the subscript of the element selected by the user from the list box when the user clicks OK. If the user cancels the dialog box by clicking Cancel, or pressing Esc or Alt+F4, it returns a number that is one less than the lower bound for subscripts in the array.

Syntax:

SelectBox (*name*, *message*, *listItems*)

name	A string expression that appears as the name of the dialog box.
message	A string expression asking the user for a response. If the length of the message exceeds the width of the list box, the message is truncated.
listItems	A one-dimensional string array whose elements become the contents of the list box. If any elements are empty, blank lines appear in the list box. It is best to avoid empty elements because the user can select a blank line from the list box.

TIP: If you want to use a line of text that is longer than the longest item in the list, add blank spaces to one of the items in the array until the list box becomes wide enough to fit beneath your text.

PopupMenu( )

SelectBox( ) Example

The following call to SelectBox( ) displays a list of applications.

```
Dim Title$
Dim Message$
Dim MyMenu$(1 To 5)
Title = "Applications"
Message = "Select an application."
MyMenu(1) = "Norton Disk Doctor"
MyMenu(2) = "Norton Speed Disk"
MyMenu(3) = "Norton Diagnostics"
MyMenu(4) = "Microsoft Word"
MyMenu(5) = "WordPerfect"
Users_Choice = SelectBox(MyMenu)
```

SelectButton

Overview See Also Example

The **SelectButton** statement simulates a mouse click on a button. The Recorder generates a **SelectButton** statement when a button is selected.

Syntax:

SelectButton *name* | *ID*

name	A string expression containing the name of the command button. A button's name is the text that appears on or is associated with it.
ID	An integer that identifies the button to select.

ActivateControl
SelectComboBoxItem
SelectListBoxItem
SetCheckBox
SetEditText
SetOption

SelectComboBoxItem

Overview See Also Example

The SelectComboBoxItem statement selects an item from a combination box. The Recorder generates a SelectComboBoxItem statement when an item is selected from a combination box.

Syntax:

SelectComboBoxItem {*name* | *ID*}, {*itemName* | *itemNum*} [, *isDoubleClick*]

name	A string expression containing the name of the combination box. Generally, this is the text in the text control visually preceding the combination box.
ID	An integer that identifies the combination box.
itemName	A string expression giving the name of the item to select.
itemNum	A numeric expression ranging from 1 to the number of lines in the combination box. It is the line number of the item to be selected.
isDoubleClick	A numeric expression that specifies whether the item is selected using a double-click or a single-click. The default is FALSE, and the item is selected using a single-click. When TRUE, selecting an item requires a double click.

ActivateControl
SelectButton
SelectListBoxItem
SetCheckBox
SetEditText
SetOption

SelectListBoxItem

Overview See Also Example

The SelectListBoxItem statement selects an item from a list box. The Recorder generates a SelectListBoxItem statement when an item is selected from a list box.

Syntax:

SelectListBoxItem {*name* | *ID*}, {*itemName* | *itemNum*} [, *isDoubleClick*]

name	A string expression containing the name of the list box. Generally, this is the text in the text control that visually precedes the list box..
ID	An integer that identifies the list box.
itemName	A string expression giving the name of the item to select.
itemNum	A numeric expression ranging from 1 to the number of lines in the list box. It is the line number of the item to be selected.
isDoubleClick	A numeric expression that specifies whether the item is selected using a double-click or a single-click. The default is FALSE, and the item is selected using a single-click. When TRUE, selecting an item requires a double click.

ActivateControl
SelectButton
SelectComboBoxItem
SetCheckBox
SetEditText
SetOption

SendKeys

See Also Example

The SendKeys statement sends keystrokes to the active application directly. It is not generated by the Recorder. The script always waits for the keys to be processed before executing the statement that follows SendKeys. However, for compatibility with other BASICs, it has a *wait* parameter that is ignored. Scripts written in other BASICs can be executed, but this parameter has no meaning.

Syntax:

SendKeys *keyStr*[, *wait*]

keyStr	A string expression containing the keystrokes to send to the active application. Keystroke Specification Format describes the format for specifying the keystrokes.
wait	A numeric expression that evaluates to TRUE or FALSE, but is always treated as though it were TRUE.

This is not a statement generated by the Recorder.

SendKeys Example

All three examples have the same functionality. Only after all the keys are sent is the next statement executed:

```
SendKeys "{PRTSC}", FALSE

SendKeys "{PRTSC}"

SendKeys "{PRTSC}", TRUE
```

SetAttr

See Also Example

The SetAttr statement changes the attributes of a file.

Syntax:

SetAttr *filename*, *fileAttr*

filename A string expression containing a complete or a relative pathname for a file. It cannot contain wildcards (? or *). An error occurs if the file does not exist.

fileAttr A numeric expression specifying the attributes to give to the file. The attributes are specified as a sum of the integers corresponding to the desired attributes:

ATTR_NORMAL	0	Normal file
ATTR_READONLY	1	Read-only file
ATTR_HIDDEN	2	Hidden file
ATTR_SYSTEM	4	System file
ATTR_ARCHIVE	32	File has changed since last backup

FileAttr( )
GetAttr( )
FileAttrSet
FileAttrGet$( )

SetAttr Example

The following example makes the AUTOEXEC.BAT file read-only and hidden:

```
SetAttr "C:\AUTOEXEC.BAT", ATTR_READONLY+ATTR_HIDDEN
```

The next example makes the AUTOEXEC.BAT file a normal file:

```
SetAttr "C:\AUTOEXEC.BAT", ATTR_NORMAL
```

SetCheckBox

Overview See Also Example

The SetCheckBox statement sets the state of a check box. The Recorder generates a SetCheckBox statement when a check boxs state changes.

Syntax:

SetCheckBox {*name* | *ID*}, *state*

name A string expression containing the name of a check box. The name is the text associated with the check box.

ID An integer that identifies the check box.

state The new state for the check box:

- If *state* is 0, the check is removed.
- If *state* is 1, the box is checked.
- If *state* is 2, the box is dimmed (only applicable for three-state check boxes).

ActivateControl
SelectButton
SelectComboBoxItem
SelectListBoxItem
SetEditText
SetOption

SetEditText

Overview See Also Example

The SetEditText statement sets the contents of a text box. The Recorder generates a SetEditText statement when the contents of a text box changes.

Syntax:

SetEditText {*name* | *ID*}, *content*

name	A string expression containing the name of a text box. Generally, this is the text in the text control that visually precedes the list box.
ID	An integer that identifies the text box.
contents	A string expression containing the new contents for the text box.

ActivateControl
SelectButton
SelectComboBoxItem
SelectListBoxItem
SetCheckBox
SetOption

SetOption

Overview See Also Example

The SetOption statement simulates a click on an option button. The Recorder generates a SetOption statement when an option button is clicked.

Syntax:

SetOption *name* | *ID*

name	A string expression containing the name of an option button. The name is the text associated with the option button.
ID	An integer that identifies the option button to click.

ActivateControl
SelectButton
SelectComboBoxItem
SelectListBoxItem
SetCheckBox
SetEditText

Sgn( )

See Also Example

The Sgn( ) function determines the sign of a specified number. It returns 1 for positive numbers, 0 for zero, and -1 for negative numbers.

Syntax:

Sgn(*exprN)*

exprN	A numeric expression whose sign is to be determined.

Abs( )
Fix( )
Int( )

Sgn( ) Example

The variable sign is assigned the sign of the specified expression.

```
sign% = Sgn(2*3/-1)
```

Shell( )

See Also Example

The Shell( ) function allows a ScriptMaker script to launch another script or an application. It is equivalent to choosing the RUN... command from the File menu in Program Manager or Norton Desktop for Windows. The ScriptMaker script can launch another ScriptMaker script or an application. The Shell( ) function returns the task ID of the command if successful. Otherwise, an error occurs.

Syntax:

Shell(*command* [, *state*])

command — A string expression that contains the name of the command to run. The name may contain a complete or relative pathname, with or without command-line options.

If the filename in the command not include a path and is not found in the current directory, the directories stored in the PATH environment variable are searched for the file to be executed.

state — A numeric expression specifying the state of the main window after execution. It can be any of the following values:

1 Normal-sized active window
2 Minimized active window
3 Maximized active window
4 Normal-sized inactive window
7 Minimized inactive window

The default state is 1, a normal-sized active window.

MCI( )
PrintFile( )

Shell( ) Example

Assuming that Notepad (NOTEPAD.EXE) is in one of the directories contained in the PATH environment variable, the following example launches Notepad in a maximized active window.

```
taskID = Shell("NOTEPAD.EXE", 3)
```

The following example launches Notepad as a normal-sized active window. Another filename is used as a command-line option. That file will appear in Notepad.

```
taskID = Shell("NOTEPAD.EXE C:\MYFILE.TXT")
```

Sin( )

See Also Example

The Sin( ) function returns the sine of the specified angle as a number of type double.

Syntax:

Sin(*angle*)

angle	A numeric expression giving the angle in radians for which to calculate the sine.

Atn( )
Cos( )
Tan( )

Sin( ) Example

The y coordinate of a point on a circle of radius 1 centered at the origin can be found by computing the sine of the angle at which the point lies on the circle.

```
'Calculate the y coordinate of the point at 30 degrees
y = Sin(30*PI/180)
```

Sleep

See Also Example

The Sleep statement makes a script wait for a specified number of milliseconds. For example, you may want to put Sleep statements between attempts to open a file that is currently in use.

Syntax:

Sleep *numMilliseconds*

numMilliseconds	The number of milliseconds to sleep.

Control Constructs

Sleep Example

In the following example, the user is prompted for a password. If the password entered is incorrect, the Sleep statement is used to introduce a delay of two seconds before the user is allowed to try again.

```
Do
   s$ = AskPassword$("Type in the password:")
   If s$ = "password" Then
      Exit Do
   End If
   Sleep 2000
Loop
```

Snapshot

See Also Example

The Snapshot statement can be used to take a snapshot of a particular section of the screen, and saves the snapshot to the clipboard. Before the snapshot is taken, each application is updated to ensure that any application in the middle of drawing has a chance to finish drawing before the snapshot is taken.

Syntax:

Snapshot [*spec*]

spec A numeric expression that specifies the portion of the screen for the snapshot. The default 0, a snapshot of the entire screen. The following are the allowable values for *spec* and descriptions of what each one specifies:

- 0 Entire screen.
- 1 Client area of the active application.
- 2 Entire window of the active application.
- 3 Client area of the active window.
- 4 Entire active window.

ClipboardClear

Snapshot Example

Using the Snapshot statement with no parameter copies the entire screen to the clipboard.

```
'Take a snapshot of the entire screen
Snapshot
```

Using a parameter specifies a particular portion of the screen. For Norton Desktop for Windows, the client area of the application is the area below the menu bar.

```
'Take a snapshot of the client area of the active application
Snapshot 1
```

Space$( )

See Also Example

The Space$( ) function returns a string containing the specified number of spaces.

Syntax:

Space$(*spaces*)

spaces	A numeric expression specifying the number of spaces to generate. The number of spaces requested cannot exceed the limits set for a strings length: 0 to 32767.

+ (concatenation) Operator
String$( )

Space$( ) Example

The following call to Space$( ) generates a string containing 100 spaces.

```
space100$ = Space$(100)
```

If you are writing records to a file, you can use the Space$( ) function to pad a field with spaces so that your string exactly fits the field. The following example finds the length of LastName and adds the proper number of blank spaces to it before writing it to a file as a field of 25 characters. For example, the last name Johnson would receive 18 spaces.

```
Length = Len(LastName)
If Length < 25 Then
   LastName = LastName + Space$(25 - Length)
End If
```

Sqr( )

See Also Example

The Sqr( ) function returns the square root of the specified number as a number of type double.

Syntax:

Sqr(*exprN*)

exprN	A numeric expression for which to calculate the square root. The number cannot be negative.

Log( )

Sqr( ) Example

The Pythagorean theorem says that the length of the hypotenuse of a right triangle is equal to the square root of the sum of the squares of the lengths of the other two sides. The following example could be used to calculate the length of the hypotenuse given the length of the other two sides.

```
's1 and s2 are the lengths of the other two sides

Function LengthOfHypotenuse#(s1#,s2#)
   LengthOfHypotenuse = Sqr(s1*s1 + s2*s2)
End Function
```

Stop

See Also Example

A script normally terminates after it executes the last line of the Main subroutine. If you need to stop execution earlier (perhaps because of an error that has occurred), you can use the **Stop** statement. It closes any open files or DDE channels before stopping the scripts execution. In addition, it displays a message that tells the line number in the script where the statement was executed. The line number is useful information when you are debugging and have more than one Stop statement in the script.

Syntax:

Stop

Control Constructs
End

Stop Example

In the following example, the user is allowed three attempts for entering the password correctly. If after three attempts, the password has not been entered correctly, the whole script is terminated using the Stop statement.

```
i% = 0
Do
   s$ = AskPassword$("Type in the password:")
   If s$ = "password" Then
      Exit Do
   End If
   i = i + 1
   If i = 3 Then
      Stop
   End If
Loop
```

StrComp( )

See Also Example

The StrComp( ) function returns an integer indicating whether the string expressions are equal or not.

0	Indicates that the string expressions are equal.
1	Indicates that *exprS1* is greater than *exprS2*.
-1	Indicates that *exprS1* is less than *exprS2*.

Syntax:

StrComp(*exprS1*, *exprS2* [, *caseSensitive*])

exprS1, *exprS2*	The string expressions to be compared.
caseSensitive	The integer 0 or 1, respectively indicating whether the comparison is case sensitive or not. The default is 0 (case sensitive).

ArraySort
String Comparison
< (less than) Operator
<= (less than or equal to) Operator
<> (not equal to) Operator
= (equal to) Operator
> (greater than) Operator
>= (greater than or equal to) Operator

StrComp( ) Example

The following example compares "apples" and "oranges". The result is -1 because the string "apples" comes before the string "oranges" in ASCII order and is, therefore, less than "oranges".

```
String1$ = "apples"
String2$ = "oranges"
Result = StrComp(String1, String2)
```

Str$( )

See Also Example

The Str$( ) function converts a numeric expression into a string. It returns a string containing the character representation of the specified number. If the expression is negative, the returned string starts with a minus sign. If it is positive, the returned string starts with a space. Converted singles have only 7 significant digits, and doubles have only 15-16.

Syntax:

Str$(*exprN*)

exprN	A numeric expression to be converted to a string.

Asc( )
Chr$( )
Hex$( )
Oct$( )
Val( )

Str$( ) Example

The following example converts the number 16 to its string equivalent.

```
strOf16$ = Str$(16)      'Result should be the string " 16"
```

String$( )

See Also Example

The String$( ) function returns a string containing a specified character repeated a specified number of times.

Syntax 1:

String$(*num*, { *charCode* | *char* })

num	A numeric expression specifying the number of characters to generate. The number of characters cannot exceed the limits set for a strings length: 0 to 32767.
charCode	A numeric expression giving the ASCII character code for the character to be repeated.
char	A string expression whose first letter is the character to be repeated.

+ (concatenation) Operator
Space$( )

String$( ) Example

The ASCII code for the letter "A" is 65. Both of the following generate a string containing 13 "A" characters.

```
stringOf13A = String$(13,65)  'Using ASCII code
stringOf13A = String$(13,"A") 'Using string
```

If you are writing records to a file, you can use String$( ) to pad a field with filler characters so that your string exactly fits the field. The next example finds the length of LastName and adds the proper number of percent signs to it. For example, the last name Johnson receives 18 characters before it is be written to a file as a field of 25 characters. If the length of Last Name is longer than the field, the Left$( ) function truncates the string.

```
Length = Len(LastName)
If Length < 25 Then
   LastName = LastName + String$(25 - Length, "%")
Else
   LastName = Left$(LastName, 25)
End If
```

String Comparison

See Also Example

String comparison is the comparison of two strings using relational operators. It is usually used to test for equality or to sort things into alphabetical order (when all characters are the same case).

When you compare strings, you compare the ASCII (American Standard Code for Information Interchange) values of both strings first characters, then their second characters, and so on. One string expression is defined as less than or greater than another based on whether or not it precedes or follows the other in ASCII order.

ASCII order, a method of representing characters in computers, sorts strings in the sequence of:

- special characters
- 0 to 9
- A to Z
- a to z

Each character is assigned a number from 0 to 127. The ASCII values from A to Z are in numeric order from 65 to 90, and the values from a to z are 97 to 122. This means. for example, that "Zebra" comes before "aardvark" and that "1500" is less than "834".

Two strings are equal if they are the same length and have exactly the same characters in exactly the same positions.

If two strings are identical up to the point where the length of one exceeds the other, the shorter string precedes the longer one. For example, "observe" precedes "observer".

If both strings are composed entirely of letters from the same case, ASCII order appears exactly like alphabetical order.

NOTE: To sort strings in alphabetical order, change all strings to uppercase or lowercase letters using the UCase$( ) and LCase$( ) functions. If you are sorting an array of strings into alphabetical order, use ArraySort, an array statement.

< Operator
<= Operator
<> Operator
= Operator
> Operator
>= Operator

String Comparison Example

All of the following examples are TRUE.

```
"alpha" = "alpha"
"alpha" > "Alpha" ' because a > A
"alpha" < "beta"        ' because a < b
"Beta" < "alpha"        ' because B < a
"a " > "a"  ' the longer string is greater than the shorter
```

All of the following examples are FALSE.

```
"alpha" <> "alpha"
"alpha" <= "Alpha"      ' because a > A
"alpha" > "beta"        ' because a < b
"Beta" > "alpha"        ' because B < a
"a " < "a"  ' the longer string is greater than the shorter
```

Sub...End Sub

See Also Example

The reserved word Sub is used to start a subroutine declaration and the reserved word pair End Sub is used to end a subroutine declaration.

You must *declare* user-defined subroutines before you can use them. In other words, the declaration of a subroutine must precede the call to that subroutine and be outside of the calling routine. The declaration contains the statements that the call executes.

Syntax:

Sub *subName* [**(** [*parameterList*] **)**]
[*localDeclarations*]
[*statements*]
End Sub

subName	Variable name for the subroutine.
parameterList	List of parameters to be passed to the subroutine.
localDeclarations	Declarations of variables to be used in the subroutine.
statements	Other statements to be used in the subroutine.

The declaration of a subroutine always ends with the **End Sub** statement. Normally the execution of the subroutine (when it is called) ends with that statement, too. However, you can abort the execution of a routine earlier by including an Exit Sub statement in the declaration. For example, if an error occurs, you may want to return to the calling routine without finishing the called routines task.

Sub...End Sub Example

The next example declares a subroutine with one parameter.

```
Sub StringPlay (LongString$)
   'A variety of statements that use LongString
End Sub
```

SystemFreeMemory( )

See Also Example

The SystemFreeMemory( ) function returns the number of bytes of free memory. The number is of type long.

Syntax:

SystemFreeMemory[( )]

SystemFreeResources( )
SystemTotalMemory( )
SystemWindowsDirectory$( )
SystemWindowsVersion$( )

SystemFreeMemory( ) Example

The following example displays the amount of free memory in a message box.

```
MsgBox "Free Memory" + Str$(SystemFreeMemory( ))
```

SystemFreeResources( )

See Also Example

The SystemFreeResources( ) function returns a number between 0 and 100 indicating the percentage of free system resources. The resources are handles to graphical objects, files, memory, and so forth.

Syntax:

SystemFreeResources[( )]

SystemFreeMemory( )
SystemTotalMemory( )
SystemWindowsDirectory$( )
SystemWindowsVersion$( )

SystemFreeResources( ) Example

The following example displays the amount of free resources in a message box.

```
MsgBox "Free Resources" + Str$(SystemFreeResources( ))
```

SystemMouseTrails

See Also Example

With the SystemMouseTrails statement, mouse trails can be turned on or off. When mouse trails is on, the mouse pointer appears to have a trail of pointers following closely behind as the pointer tracks across the screen. This is a useful feature for computers with LCD monitors because it makes the pointer easier to follow.

Syntax:

SystemMouseTrails *state*

state	A numeric expression that turns mouse trails off if it evaluates to 0. Any value besides 0 turns mouse trails on. The value of *state* does not change the length of the trail.

NOTE: Each time the SystemMouseTrails statement is executed, the new state of mouse trails is saved to the WIN.INI file.

ReadINI$( )
ReadINISection
WriteINI

SystemMouseTrails Example

The following examples shows the SystemMouseTrails statement turning mouse trails on and off.

```
'Turn mouse trails off
SystemMouseTrails 0
'Turn mouse trails on
SystemMouseTrails 1
'Also turns mouse trails on
SystemMouseTrails -2
```

SystemRestart

See Also Example

If at any time you want to restart Windows from within a script, the SystemRestart statement can be used.

Syntax:

SystemRestart

CAUTION: When the SystemRestart statement is executed, Windows restarts immediately. Take precautions when using this statement because all other applications are also aborted and may result in loss of data. You may want to give the user a warning and a chance to save any data in other applications before executing the SystemRestart statement.

Shell( )

SystemRestart Example

The following example displays a message box for the user. The message asks whether to restart Windows or not. If the answer is yes, the SystemRestart statement is executed.

```
'Propose the question
Answer% = MsgBox("Do you really want to restart Windows?", 4+32)

'Was the answer yes?
If Answer = 6 Then
   SystemRestart  'The answer was yes, so restart
End If
```

SystemTotalMemory( )

See Also Example

The SystemTotalMemory( ) function returns the total number of bytes of available memory in Windows. The number is of type long.

Syntax:

SystemTotalMemory[( )]

SystemFreeMemory( )
SystemFreeResources( )
SystemWindowsDirectory$( )
SystemWindowsVersion$( )

SystemTotalMemory( ) Example

The following example displays the total amount of available memory in Windows.

```
MsgBox "Total Memory" + Str$(SystemTotalMemory( ))
```

SystemWindowsDirectory$( )

See Also Example

The **SystemWindowsDirectory$( )** function returns a string containing the full pathname of the Windows directory (for example, "C:\WINDOWS").

Syntax:

SystemWindowsDirectory$[( )]

SystemFreeMemory( )
SystemFreeResources( )
SystemTotalMemory( )
SystemWindowsVersion$( )

SystemWindowsDirectory$( ) Example

The following example displays the windows directory in a message box:

```
MsgBox "Windows Directory:" + SystemWindowsDirectory$( )
```

SystemWindowsVersion$( )

See Also Example

The SystemWindowsVersion$( ) function returns a string containing the Windows version number (for example, "3.0" or "3.10").

Syntax:

SystemWindowsVersion$[( )]

SystemFreeMemory( )
SystemFreeResources( )
SystemTotalMemory( )
SystemWindowsDirectory$( )

SystemWindowsVersion$( ) Example

The following example displays the windows version in a message box:

```
MsgBox "Windows Version:" + SystemWindowsVersion$( )
```

Tan( )

See Also Example

The Tan( ) function returns the tangent of the specified angle as a number of type double.

Syntax:

Tan(*angle*)

angle	A numeric expression giving the angle in radians of which to calculate the tangent.

Atn( )
Cos( )
Sin( )

Tan( ) Example

The tangent of an angle is equal to the sine of an angle divided the angle's cosine. The following example should confirm this.

```
'Calculate the tangent of 30 degrees
tan30 = Tan(30*PI/180)

'The result of the previous calculation should equal
' the result of the following calculation
sin30_cos30 = Sin(30*PI/180)/Cos(30*PI/180)
```

Text

Overview See Also Example

The Text statement defines a dialog box control that displays text. The text control can be used to name list boxes, combination boxes, and text boxes, all of which do not have their own names.

Syntax:

Text *x, y, width, height*, *name*

x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in dialog units. The upper-left corner of the window is 0, 0.
width, *height*	The integer expressions indicating the width and height of the dialog box in dialog units.
name	String variable or literal containing text. When text is associated with a text box, list box, or combination box, *name* can be the name of the box and contain an ampersand & in front of the character to be used as an accelerator key.

Begin Dialog...End Dialog
Dialog
Dialog( )

Dialog, Text, and TextBox Example

The following script displays a dialog box containing a text control named Serial Number and a text box control for entering a serial number. When the Dialog statement ends, a message box displays the serial number that was entered.

```
Sub Main( )

   Begin Dialog SerialNumDialog 16,32,110,33, "Serial Number"
      Text 5,6,57,8, "Serial Number:"
      TextBox 5,15,51,12, .SerialNumber
      OKButton 64,13,41,14
   End Dialog

   Dim SerialNumDialog1 As SerialNumDialog

   Dialog SerialNumDialog1

   'Display the entered serial number
   MsgBox SerialNumDialog1.SerialNumber

End Sub
```

TextBox

Overview See Also Example

The TextBox statement defines a text box that appears within a dialog box template.

Syntax:

TextBox *x, y, width, height*, *.field*

x, *y*	The integer expressions indicating the horizontal and vertical distances from the upper-left corner of the window to the upper-left corner of the dialog box in dialog units. The upper-left corner of the window is 0, 0.
width, *height*	The integer expressions indicating the width and height of the dialog box in dialog units.
.field	A string variable used to set and/or retrieve value of text box. Setting this field gives the text box an initial value.

Begin Dialog...End Dialog
Dialog
Dialog( )

Time$

See Also Example

To the Time$ statement assigns a new time to the system time.

Syntax:

Time$ = *newTime*

The *newTime* parameter can be specified as a string in any of the three following formats:

- HH
- HH:MM
- HH:MM:SS

When setting the time, you never need to precede single digit hours, minutes, or seconds with a zero.

Date$
Date$( )
Now( )
Time$( )
Timer( )

Time$ Example

The following examples illustrate the use of **Time$** to set the time.

```
Time$ = "16:05"    'Set the time to 16:05:00 or 4:05 p.m.
Time$ = "8:5"      'Set the time to 8:05:00 a.m.
```

Time$( )

See Also Example

The **Time$( )** function returns the system time as a string in the format HH:MM:SS, where HH is the hour, MM is the minutes, and SS is the seconds. A 24-hour clock is used. If the hour can be displayed as a single digit (for example, 0 through 9), no zero precedes the single digit. But a zero does precede the minute and second if the corresponding value is only a single digit.

Syntax:

Time$[( )]

Date$
Date$( )
Now( )
Time$
Timer( )
TimeValue( )

Time$( ) Example

The following example saves the current system time as the variable currentTime.

```
currentTime$ = Time$( )
```

Timer( )

See Also Example

The Timer( ) function returns the number of seconds since midnight. The number is of type long.

Syntax:

Timer[( )]

Date$
Date$( )
Now( )
Time$
Time$( )

Timer( ) Example

The following example stores the elapsed seconds since midnight in a variable of type long.

```
secSinceMidnight& = Timer( )
```

TimeSerial( )

See Also Example

The TimeSerial( ) function returns the serial time, a number of type double, representing the specified hour, minute, and second. The zero date (Dec. 20, 1899) is also included in the returned serial number.

Syntax:

TimeSerial(*hour*, *minute*, *second*)

hour	A numeric expression from 0 to 24 giving the hour to be encoded in the serial time.
minute	A numeric expression from 0 to 60 giving the minute to be encoded in the serial time.
second	A numeric expression from 0 to 60 giving the second to be encoded in the serial time.

Date and Time Calculations
DateSerial( )
DateValue( )
Day( )
Hour( )
Minute( )
Month( )
Now( )
Second( )
TimeValue( )
Weekday( )
Year( )

TimeSerial( ) Example

To obtain the serial time for 3:30 PM:

```
serialDT# = TimeSerial(15,30,0)
```

TimeValue( )

See Also Example

The TimeValue( ) function converts the specified string expression to a serial time. It returns the serial time as a number of type double.

Syntax:

TimeValue(*timeStr*)

timeStr	A string expression containing a valid time specification. The ordering of the time items must follow the current settings in use by Windows and specified in the INTL section of the WIN.INI file. The time items may be separated by time separators such as colon (:) or period (.). If a particular time item is missing, then the missing time items are set to zero. For example "10 pm" would be interpreted as "22:00:00". Including a date in the string is optional. If no date is included, the zero date (Dec. 20, 1899) is assumed.

Date and Time Calculations
DateSerial( )
DateValue( )
Day( )
Hour( )
Minute( )
Month( )
Now( )
Second( )
Time$( )
TimeSerial( )
Weekday( )
Year( )

TimeValue( ) Example

To obtain the serial time for 3:30 PM:

```
serialDT# = TimeValue("3:30 PM")
```

Trim$( )

See Also Example

The Trim( ) function returns the specified string with any leading and/or trailing spaces removed.

Syntax:

Trim$(*exprS*)

exprS A string expression.

Left$( )
LTrim$( )
RTrim$( )
Mid$( )
Right$( )

Trim$( ) Example

The following examples use Trim$( ) to delete leading and trailing spaces.

```
aString$ = "  3 leading and 3 trailing spaces  "

'Now remove the leading and trailing spaces
aString = Trim$(aString)
'aString should now be equal to the string
'"3 leading and 3 trailing spaces"
```

TRUE

See Also Example

TRUE is a numeric constant with a value of -1. It is used in relational expression and logical expression.

Conditional Constructs
FALSE

TRUE Example

The following example returns the value TRUE if a specified integer is even. Otherwise, the function returns the value FALSE.

```
Function Even(n As Integer)
   If (n MOD 2) = 0 Then
      Even = TRUE
   Else
      Even = FALSE
   End If
End Function
```

TYPE_DOS

See Also Example

TYPE_DOS is a numeric constant with a value of 1. It is used as a return value from the FileType( ) function to indicate that a specified file is a DOS application.

TYPE_WINDOWS
FileType( )

TYPE_WINDOWS

See Also Example

TYPE_WINDOWS is a numeric constant with a value of 2. It is used as a return value from the FileType( ) function to indicate that a specified file is a Windows application.

TYPE_DOS
FileType( )

UBound( )

See Also Example

The UBound( ) function returns an integer indicating the highest subscript number for the specified dimension of the specified array.

Syntax:

UBound(*arrayName* [, *dimension*])

arrayName	The name of the array.
dimension	The dimension. The default is 1 for the first dimension.

NOTE: If the UBound( ) function is used on an array with no dimensions, then a run-time error will occur. The ArrayDims( ) function can be used to first check if the array has any dimensions.

ArrayDims
ArraySort
Dim
LBound( )
Option Base
ReDim

UBound( ) Example

The following example finds the upper bound for subscripts in the first dimension of a two-dimensional array using the UBound( ) function.

```
Dim Array1(0 To 3, 0 To 2) As Integer

'Determine the upper bound
highest_subscript = UBound(Array1)
```

UCase$( )

See Also Example

The UCase$( ) function converts lowercase letters of a string to uppercase. It returns a string containing all the characters of the specified string in uppercase.

Syntax:

UCase$(*exprS*)

exprS	A string expression that is to be converted to uppercase.

LCase$( )

UCase$( ) Example

The following example results in the string "THIS IS ONLY A TEST" being assigned to the variable newString.

```
newString$ = UCase$("This is Only a Test!")
```

User-Defined Dialog Boxes Overview

See Also

In a ScriptMaker script, you can define your own dialog boxes to provide output to and to receive input from the user. To display a dialog box other than the predefined dialog boxes, you must:

- Create a template for the user-defined dialog box in the Dialog Editor. Each *template* defines a dialog box's size, its controls (such as command buttons and text boxes), the size and position of those controls, and so forth.
- Insert the template for that dialog box in your program as the declaration of the dialog box. Each control of the template has a line that defines it within the construct.
- Add statements to the script to:
  - Declare any variables that appear in the template.
  - Display an instance of the dialog box template.
  - Preset the controls of the dialog box for the user.
  - Retrieve the information the user enters in the dialog box.

The steps from creating to displaying a dialog box template:

- Creating a New Dialog Box
- Copying a Template into a Script
- Inserting a Dialog Box Into a Script

The function and statement that display the dialog box:

- Dialog( )
- Dialog

The statements in a dialog box template declaration:

- Begin Dialog...End Dialog
- CancelButton
- CheckBox
- ComboBox
- GroupBox
- ListBox
- OKButton
- OptionButton
- OptionGroup
- PushButton
- Text
- TextBox

The easy way out:

- Predefined Dialog Boxes Overview

User-defined Functions and Subroutines

See Also Example

User-defined functions and subroutines are control constructs that:

- Allow statements that are repeated in various parts of the script to be written only once.
- Make some variables invisible to other parts of the script, out of the scope of any other function or subroutine. These variables are not changed by any other function or subroutine unless they are passed to it for that purpose.
- The script is easier to understand, debug, and maintain.

With the exception of the Main subroutine that automatically begins execution when the script runs, all functions and subroutines are called by another function or subroutine. The one making the call is the calling routine and the one being called is the called routine. When a calling routine makes a call, its execution is put on hold until the called routine has been executed. Then execution of the calling routine continues with the statement after the call.

A script can have an unlimited number of user-defined functions and subroutines.

User-Defined Functions and Subroutines Example

The following user-defined function can be used to determine whether a specified positive number is a prime number.

```
Function IsPrime(ByVal n As Integer) As Integer
   For i = 2 To n - 1
      If n MOD i = 0 Then
            IsPrime = FALSE
            Exit Function
      End If
   Next

   IsPrime = TRUE
End Function
```

Val( )

See Also Example

The Val( ) function converts a string containing numeric characters to a number of type double. The numeric characters in the string can be the digits 0 to 9, a leading minus sign, a hexadecimal number in the format &H*HexDigits*, an octal number in the format &O*OctalDigits*. Spaces, tabs, and linefeeds are ignored. The function stops reading characters from the string when it finds a nonnumeric character that it does not ignore.

Syntax:

Val(*exprS*)

exprS	A string expression containing a string representation of a number that is to be converted to its numeric equivalent.

Asc( )
Chr$( )
Hex$( )
Oct$( )
Str$( )

Val( ) Example

The following three lines convert hexadecimal, octal, and decimal string representations of the number 16 into their numeric equivalents:

```
hexConv% = Val("&H10")   'hexConv should equal 16
octConv% = Val("&O20")   'octConv should equal 16
decConv% = Val("16")     'decConv should equal 16
```

ViewportClear

See Also Example

The ViewportClear statement clears the open viewport window. The buffer is erased and the window is cleared.

Syntax:

ViewportClear

Print
ViewportClose
ViewportOpen

Print, ViewportClear, and ViewportOpen Example

The following script opens a viewport window in the upper-left corner of the screen and continuously displays the current time. When the time is updated, the window is cleared so that the new time always appears at the beginning of the window.

```
Sub Main( )
   Dim lastTime$

   ViewportOpen "Time", 0, 0, 100, 70

   Do
      If lastTime$ <> Time$( ) Then
            ViewportClear
            lastTime$ = Time$( )
            Print Time$( )
      End If
   Loop
End Sub
```

ViewportClose

See Also Example

The ViewportClose statement closes an open viewport window.

Syntax:

ViewportClose

NOTE: Only the script that originally opened the viewport window can close it. A call to ViewportClose by a script that called ViewportOpen when a viewport window was already open has no effect.

Print
ViewportClear
ViewportOpen

ViewportClose Example

In the following example, a viewport window appears and displays a message for ten seconds. Then the viewport closes.

```
ViewportOpen
Print "Here is a message!"
Sleep 10000
ViewportClose
```

ViewportOpen

See Also Example

You must create a viewport window before text can be output to it. The ViewportOpen statement creates a viewport window.

Syntax:

ViewportOpen [*name* [, *x*, *y* [, *width*, *height*]]]

name	The name that is to appear in the viewport window's caption. If *name* is omitted, then the default caption is "ScriptMaker Viewport."
x, *y*	The integer expressions in twips for the horizontal and vertical locations of the viewport window relative to the upper-left corner of the screen (which is 0, 0). There are 1440 twips in an inch. The default is slightly off center.
width, *height*	The integer expressions for the width and height of the viewport window in dialog units.

NOTE: Only one viewport window can be open at any one time. Any scripts executing at the same time that output text to a viewport window, output their text to the same viewport window. If a viewport window is currently open, additional ViewportOpen statements have no effect.

When the application that opened the viewport window terminates, the viewport window automatically closes.

Since the viewport's buffer size is 32-kilobytes, not all of the information may be visible at once. The vertical and horizontal scroll bars allow the user to scroll to different parts of the buffer using the mouse. The following keystrokes also scroll through the buffer.

Scroll Action	Keystroke
Up by one line.	Up
Down by one line.	Down
Left by one column.	Left
Right by one column.	Right
To the first line.	Home
To the last line.	End
Up by one page.	PgUp
Down by one page.	PgDn
Left by one page.	Ctrl+PgUp
Right by one page.	Ctrl+PgDn

Print
ViewportClear
ViewportClose

VK_LBUTTON

See Also Example

VK_LBUTTON is a numeric constant with the value 1. It is used with the QueMouse statements to indicate that a mouse action uses the left mouse button.

QueMouseClick
QueMouseDblClk
QueMouseDblDn
QueMouseDn
QueMouseUp
VK_RBUTTON

VK_LBUTTON Example

The following example simulates a mouse click using the left mouse button.

```
'Left mouse button click at (x=167, y=205)
QueMouseClick VK_LBUTTON, 167, 205

'Play the click
QueFlush TRUE
```

VK_RBUTTON

See Also Example

VK_RBUTTON is a numeric constant with the value 2. It is used with the QueMouse statements to indicate that a mouse action uses the right mouse button.

QueMouseClick
QueMouseDblClk
QueMouseDblDn
QueMouseDn
QueMouseUp
VK_LBUTTON

VK_RBUTTON Example

The following example performs a double-click using the right mouse button.

```
'Right mouse double-click at (x=100, y=101)
QueMouseDblClk VK_RBUTTON, 100, 101

'Play the double-click
QueFlush TRUE
```

VLine

See Also Example

VLine scrolls up or down a specified number of lines in an active window's viewport. During a recording session with the Recorder, clicking the arrow button at either end of a vertical scroll bar generates a VLine statement.

Syntax:

VLine [*lines*]

lines Number of lines to scroll. If positive, the direction of scrolling is down. If negative, the direction of scrolling is up. The default is to scroll down one line.

AppActivate
HLine
HPage
HScroll
VPage
VScroll
WinActivate

VLine Example

Scroll up one line using the vertical scroll bar:

```
VLine -1
```

VPage

See Also Example

VPage scrolls up or down by a specified number of pages in an active window's viewport. During a recording session with the Recorder, clicking in the scroll area on either side of the scroll box generates a VPage statement.

Syntax:

VPage [*pages*]

pages Number of pages to scroll. If positive, the direction of scrolling is down. If negative, the direction of scrolling is up. The default is to scroll down one page.

AppActivate
HLine
HPage
HScroll
VLine
VScroll
WinActivate

VPage Example

Scroll down one page using the vertical scroll bar:

```
VPage 1
```

VScroll

See Also Example

The VScroll statement positions the scroll box a percentage of the way down the total range of a vertical scroll bar in the active window's viewport. During a recording session with the Recorder, dragging the scroll box to a new position within the scroll bar generates a VScroll statement.

Syntax:

VScroll *percentage*

percentage	An integer specifying a percentage of the scroll bar, and, therefore, the location at which to place the scroll box.

AppActivate
HLine
HPage
HScroll
VLine
VPage
WinActivate

VScroll Example

Set the vertical scroll box at the very end of the scroll bar:

```
VScroll 100
```

WaitForKey( )

See Also Example

The WaitForKey( ) function suspends script processing until any one of five keys is pressed, and returns an integer indicating the position within the parameter list of the key that was pressed.

Syntax:

WaitForKey(*char*, *char*, *char*, *char*, *char*)

char A string expression representing a keystroke, or an empty string (""). Valid values are the letters, the numbers 1 through 9, and the following special key designations:
{BACKSPACE} or {BS}
{DELETE} or {DEL}
{END}
{ENTER}
{ESCAPE} or {ESC}
{F1} through {F16}
{HOME}
{INSERT}
{PGDN}
{PGUP}
{SPACE} or {SP}
{TAB}

This function requires five parameters; empty strings ("") should be used to define fewer than five keys. The integer returned by this function indicates which *char* key was pressed; for example, if "c" is the third *char* parameter and the user presses **c**, then WaitForKey returns 3. The *char* parameter is case-insensitive, so "A" and "a" are equivalent parameters. However, WaitForKey itself responds only to lowercase letters; if the user holds down the Shift key while pressing a second key, the second key is ignored. (Note that the Shift key is not a valid value for *char*.)

This function is ideal for demos, for quickly achieving functionality similar to option buttons, or for preventing a user from continuing script processing unless the user knows what key to press.

Sleep

WaitForKey( ) Example

The following example uses the MsgBox statement to instruct the user on how to proceed with the script, and then uses WaitForKey( ) to check the user's response. If the user presses **s**, the Solitaire application starts; if the user presses **x**, the script terminates. Script processing is suspended until one or the other of these keys is pressed.

```
crlf$ = Chr$(13)+Chr$(10)
a$ = "Press OK, then..." + crlf + crlf
b$ = "Press s to play Solitaire." + crlf
c$ = "Press x to exit this script now."
d$ = a + b + c
MsgBox d
key = WaitForKey("s", "x", "", "", "")
If key = 1 Then
   taskID = Shell("c:\windows\sol.exe")
Else End
End If
```

WBTVersion$( )

See Also Example

The WBTVersion$( ) function returns a string containing the version number for the files providing the Batch Runner extensions to the ScriptMaker language. (Batch Runner is a Windows batch language included with earlier versions of Norton Desktop for Windows.)

Syntax:

WBTVersion$ [( )]

ScriptMakerHomeDir$( )
ScriptMakerOS( )
ScriptMakerVersion$( )

WBTVersion$( ) Example

The following example uses the MsgBox statement to display the string returned by WBTVersion$( ).

```
message$ = "Batch Runner Extensions: Version "
MsgBox message + WBTversion$()
```

Weekday( )

See Also Example

The Weekday( ) function returns a number from 1 to 7 that represents the day of the week extracted from a serial date. The weekday 1 is Sunday.

Syntax:

Weekday(*serialDateTime*)

serialDateTime	Serial date, a number of type double, from which the weekday is to be extracted.

DateSerial( )
DateValue( )
Day( )
Hour( )
Minute( )
Month( )
Now( )
Second( )
TimeSerial( )
TimeValue( )
Year( )

While...Wend

See Also Example

A conditional loop terminates when the value of the logical expression that controls it changes. A While loop checks the logical expression before it executes any of the statements inside the loop. If the expression is true, the statements are executed. When the expression becomes false, the loop terminates.

Syntax:

While *exprL*

[*statement*]...

Wend

exprL	A relational or logical expression.
statement	An executable statement.

At least one statement inside the loop must change the value of the logical expression for every iteration, or the loop becomes an infinite loop and never terminates. Use <= or >= to avoid the infinite loop that would result if the operator was = and the ending value was accidentally bypassed.

Do...Loop
For...Next
Looping Constructs

While...Wend Example

The following example calculates the factorial of an integer greater than zero. In this example, the statements in the Do...Loop execute at least once. If you are sure that you have to execute the statements inside the loop at least once, you can use a Do loop with the expression at the end. The loop terminates when the input FactNum is greater than or equal to 0. The While loop terminates when the counter is greater than FactNum. The value of FactNum in the first loop and the value of Counter in the second change during every iteration of the loop.

```
Sub FactCal
Dim Counter As Integer  ' For loop counter.
Dim Factorial As Integer      ' Stores the result of factorial.
Dim FactNum As Integer  ' Input number for calculation.

Do
   '  get number greater than zero
   FactNum = Val(InputBox$ ("Enter a positive integer."))
   If FactNum <= 0 Then
      MsgBox "Try again"
   End If
Loop Until FactNum > 0  ' get a positive integer.
' Now FactNum is greater than or equal to zero. Initialization:
Factorial   = 1
Counter     = 1

' Calculate factorial. When Counter is greater than
' FactNum, the While loop terminates.
While Counter <= FactNum
   Factorial = Factorial * Counter
   Counter = Counter + 1
Wend
MsgBox "The factorial is:  " + Str$(Factorial) + "."
End Sub
```

WinActivate

Overview See Also Example

The WinActivate statement can activate any window, dialog box, command button, check box, and option button using the specified name or handle. It can activate a text box, list box, or combination box using the specified handle.

During a recording session with the Recorder, activating an application generates a WinActivate statement.

Syntax:

WinActivate {*name* | *handle*}

name	A string expression containing the name, partial name, or parent/child hierarchy for a window, dialog box, command button, check box, or option button. Full name: "No T&ext" Partial Name:"Admin" for "Norton Administrator" or "No T" for "No T&ext" The hierarchy is shown by separating names with a vertical bar (|): "Norton Desktop|File Open" The default is the active window.
handle	A numeric expression for the handle to the window, which can be obtained using WinFind( ) or WinList.

NOTE: The ampersand (&) used to create accelerator keys is considered a character in the name. For example, the No Text option button's name is "No T&ext".
The parent/child hierarchy cannot be used to locate a modal dialog box or one of its children because a modal dialog box disables its parent, making the parent inaccessible to WinActivate and other statements.

AppActivate
AppClose
WinClose
WinFind( )
WinList

WinActivate Example

The following example activates the Norton Desktop for Windows.

```
WinActivate "Norton Desktop"
```

The following example activates the No Text option button in the Toolbar Configuration dialog box in Norton Administrator. This example shows a full name (Norton Administrator), two partial names ("Toolbar" and "No T"), and the parent/child hierarchy from the Norton Administrator console to the No Text option button.

```
WinActivate "Norton Administrator|Toolbar|No T"
```

WinClose

Overview See Also Example

The WinClose statement closes the specified window, dialog box, or dialog-box control.

Syntax:

WinClose [*name*]

name	A string expression containing the name, partial name, or parent/child hierarchy for a window, dialog box, command button, check box, or option button. Full name: "No T&ext" Partial Name:"Admin" for "Norton Administrator" or "No T" for "No T&ext" The hierarchy is shown by separating names with a vertical bar (|): "Norton Desktop|File Open" The default is the active window.

CAUTION: In some applications, closing a control causes unexpected and unwanted side effects.

AppActivate
AppClose
WinActivate

WinClose Example

The following example closes an application matching the partial name "word".

```
WinClose "word"
```

The next example makes the No Text option button disappear from the Toolbar Configuration dialog box temporarily.

```
WinClose "Norton Administrator|toolbar|no T"
```

Window Overview

See Also

Once a Windows application is running, ScriptMaker gives you complete control over all its windows. ScriptMaker has two sets of statements and functions that control windows: One set starts with App (such as AppActivate) and the other set starts with Win (as in WinActivate). The App set control only top-level main windows (such as "Norton Desktop for Windows", "Microsoft Word", or "Microsoft Word - Document2"). If you know the complete name of the main window you wish to manipulate, you can use ScriptMaker statements to start, activate, move, resize, get information about, or close its window. The App set takes only the complete name of a top-level main window as a parameter with the exception of AppFind$( ) which uses a partial name and returns the complete name. For the rest of the App statements and functions, a run-time error occurs if the name is not the exact name of a top-level main window that is open and active. If a modal dialog box from the application is open, the window is disabled and cannot be recognized by App statements and functions. The default name is the name of the active main window.

The Win set can control any windows, dialog boxes, and their controls that would be visible to the user. (This is because, technically speaking, Windows considers all windows, dialog boxes, and controls to be windows.) For example, a check box can be controlled if it exists, is in an open dialog box, and be enabled (not dimmed).The Win set takes a string expression containing the name, partial name, or parent/child hierarchy for a window, dialog box, command button, check box, or option button as a parameter. Some examples are:

Full name: "No T&ext" is the full name of the No Text option button. The ampersand (&) used to create the accelerator key (the underlined character) is considered a character in the name.
Partial name: "Admin" for "Norton Administrator" or "No T" for "No T&ext". Use unambiguous partial names.
Parent/child hierarchy: The hierarchy is shown by separating names with a vertical bar (|): "Norton Desktop|File Open". The parent/child hierarchy cannot be used to locate a modal dialog box or one of its children because a modal dialog box disables its parent, making the parent inaccessible.
The default name is the name of the active main window or dialog box.

The Win set cannot identify text boxes, list boxes, and combination boxes by name. The static text closest to a text box, list box, or combination box is usually considered the name of that control, but the Win statements and functions do not recognize this.
These last three types of controls can be identified only by their handles. Of the Win set, only the WinActivate statement accepts a handle as a parameter.
While the WinFind( ) function returns a handle and the WinList statement returns an array of handles for top-level windows, you must supply a complete or partial name. This means you cannot find the handles for text boxes, list boxes, or combination boxes using ScriptMaker. Names of these controls are not understood by the Win set. However, if you have the Windows Software Development Kit (SDK), you can use tools like SPY to find the handles.
Only the WinActivate statement can be generated during a recording session with the Recorder. For example, activating an application generates a WinActivate statement. Despite the fact that the WinMove statement can be used to move a dialog box, no dialog box movements are recorder in a macro. However, you can add a WinMove statement to the macro manually.

Dialog-box Controls Overview
Menu Overview

The **App** set:

AppActivate
AppClose
AppFileName$( )
AppFind$( )
AppGetActive$( )
AppGetPosition
AppGetState
AppHide
AppList
AppMaximize
AppMinimize
AppMove
AppRestore
AppSetState
AppShow
AppSize
AppType( )

The **Win** set:

WinActivate
WinClose
WinFind( )
WinList
WinMaximize
WinMinimize
WinMove
WinRestore
WinSize

The statements that manipulate Horizontal and Vertical scroll bars:

HLine
HPage
HScroll
VLine
VPage
VScroll

WinFind( )

Overview See Also Example

The WinFind( ) function returns an integer that is the handle for the specified window, dialog box, or control. If nothing matches the specified partial name, the function returns 0. If more than one windows title matches the string you specify, the function returns the handle for the matching window that was most recently used or started.

Syntax:

WinFind(*name*)

name	A string expression containing the name, partial name, or parent/child hierarchy for a window, dialog box, command button, check box, or option button. Full name: "No T&ext" Partial Name:"Admin" for "Norton Administrator" or "No T" for "No T&ext" The hierarchy is shown by separating names with a vertical bar (|): "Norton Desktop|File Open"

The WinActivate statement is the only window statement that will take either a window handle or a window name for specifying the window.

AppFind$( )
AppGetActive$( )
AppList
WinList

WinFind( ) Example

In the following example, if the specified window is found, it is activated:

```
handle% = WinFind("word")

'If such a window is found, Then activate it with WinActivate
If handle <> 0 Then
   WinActivate handle
End If
```

If you know that the window exists and is enabled, you can use

```
handle% = WinFind("word")
```

WinList

Overview See Also Example

The WinList statement obtains the handles to all the main windows.

Syntax:

WinList *handlesArray*

handlesArray	A one-dimensional integer array that will hold the handles to the top-level windows. The array is automatically resized to hold the handles. After the call, use the **LBound( )** and **UBound( )** functions to determine the new bounds of the array, and therefore the number of windows found.

AppFind$( )
AppGetActive$( )
AppList
WinFind( )

WinList Example

In the following example, the handles to all of the main windows and their names are obtained, and then displayed in a message box:

```
Dim winhandles%( ), handlelist$, appnames$( )

'Get the handles and names
WinList winhandles
AppList appnames

crlf$ = Chr$(13) + Chr$(10)
tab$ = Chr$(9)
numfound% = UBound(winhandles) - LBound(winhandles) + 1
handlelist$ = Str$(numfound) + " windows found:" + crlf + crlf
For i = LBound(winhandles) To UBound(winhandles)
   handlelist = handlelist + Str$(winhandles(i)) + tab
   handlelist = handlelist + appnames(i) + crlf
Next

'Display the list
MsgBox handlelist
```

WinMaximize

Overview See Also Example

The WinMaximize statement maximizes the specified window. Otherwise the statement has no effect. A run-time error occurs if the specified window does not exist.

Syntax:

WinMaximize [*name*]

name	A string expression containing the name, partial name, or parent/child hierarchy for a window, dialog box, command button, check box, or option button. Full name: "No T&ext" Partial Name:"Admin" for "Norton Administrator" or "No T" for "No T&ext" The hierarchy is shown by separating names with a vertical bar (|): "Norton Desktop|File Open" The default is the active window.

WinActivate
WinMinimize
WinMove
WinRestore
WinSize

WinMaximize Example

The following examples maximize and then restore the Microsoft Word - Document1 window.

```
WinMaximize "Document1"
...
WinRestore "Document1"
```

WinMinimize

Overview See Also Example

The WinMinimize statement minimizes the specified window. Otherwise the statement has no effect. A run-time error occurs if the specified window does not exist.

Syntax:

WinMinimize [*name*]

name A string expression containing the name, partial name, or parent/child hierarchy for a window, dialog box, command button, check box, or option button.

Full name: "No T&ext"

Partial Name:"Admin" for "Norton Administrator" or "No T" for "No T&ext"

The hierarchy is shown by separating names with a vertical bar (|): "Norton Desktop|File Open"
The default is the active window.

WinActivate
WinMaximize
WinMove
WinRestore
WinSize

WinMinimize Example

The following examples minimizes the Norton Desktop window.

```
WinMinimize "Desktop"
```

WinMove

Overview See Also Example

The WinMove statement moves the specified window, dialog box, or dialog-box control to the specified location. A run-time error occurs in the specified window does not exist.

Syntax:

WinMove x, y[, *name*]

x, *y*	The numeric expressions indicating the horizontal and vertical distances in pixels from the upper-left corner of the window to the upper-left corner of the dialog box. The upper-left corner of the window is 0, 0.
name	A string expression containing the name, partial name, or parent/child hierarchy for a window, dialog box, command button, check box, or option button. Full name: "No T&ext" Partial Name:"Admin" for "Norton Administrator" or "No T" for "No T&ext" The hierarchy is shown by separating names with a vertical bar (|): "Norton Desktop|File Open" The default is the active window.

WinActivate
WinMaximize
WinMinimize
WinRestore
WinSize

WinMove Example

The following example moves the LAN Inventory window to the specified location on the Norton Administrator console.

```
WinMove 20, 20, "Norton Administrator|LAN Inventory"
```

The next statement moves the Filter button on the LAN Inventory window to the upper-left corner of the window (on top of the name of the first column in the window).

```
WinMove 0, 0 "Norton Administrator|LAN Inventory|Filter"
```

WinRestore

Overview See Also Example

The WinRestore statement restores the specified window if it is currently minimized or maximized. Otherwise the statement has no effect. A run-time error occurs in the specified window does not exist.

Syntax:

WinRestore [*name*]

name A string expression containing the name, partial name, or parent/child hierarchy for a window, dialog box, command button, check box, or option button.

Full name: "No T&ext"

Partial Name:"Admin" for "Norton Administrator" or "No T" for "No T&ext"

The hierarchy is shown by separating names with a vertical bar (|): "Norton Desktop|File Open"
The default is the active window.

WinActivate
WinMaximize
WinMinimize
WinMove
WinSize

WinSize

Overview See Also Example

The WinSize statement resizes the specified window, dialog box, or dialog-box control. A run-time error occurs in the specified window does not exist.

Syntax:

WinSize *width*, *height* [, *name*]

width, *height*	Integer expressions for the new width and height in pixels.
name	A string expression containing the name, partial name, or parent/child hierarchy for a window, dialog box, command button, check box, or option button. Full name: "No T&ext" Partial Name:"Admin" for "Norton Administrator" or "No T" for "No T&ext" The hierarchy is shown by separating names with a vertical bar (|): "Norton Desktop|File Open" The default is the active window.

WinActivate
WinMaximize
WinMinimize
WinMove
WinRestore

WinSize Example

The following resizes a Norton Administrator chart window.

```
WinSize 300, 300, "norton admin|2-D Column"
```

The following example resizes the Copy button in that chart window.

```
WinSize 35, 35, "norton admin|2-D Column|copy"
```

Word$( )

See Also Example

The Word$( ) function returns a string containing all the words from a specified string starting with the first word specified and ending with the last word specified. Words are delimited by spaces, tabs, and carriage-return/linefeeds.

Syntax:

Word$(*text*, *first*[, *last*])

text	A string expression containing the text to parse.
first	A numeric expression specifying the first word to retrieve. Word 1 is the first word of the text.
last	A numeric expression specifying the last word to retrieve. The default is the value for first so only one word is returned.

If the number of words to retrieve is greater than one, then all the separators (spaces, tabs, and carriage-return/linefeeds) separating the retrieved words are also included in the returned string.

If *first* is greater than the number of words in *text*, an empty string is returned.

If *last* is greater than the number of lines in *text*, then all lines from *first* to the end of *text* are returned.

Item$( )
ItemCount( )
Line$( )
LineCount( )
WordCount( )

Word$( ) and WordCount( ) Example

In the following example, the string variable whoIsIt is parsed for a first, middle, and last name. If the string has more than two words, the first word is the first name, the second word is the middle name, and the third word is the last name. If the string has two words, the first word is the first name and the second word is the last name. If neither of the previous two conditions holds, the first word is the first name.

```
Dim first$, middle$, last$, whoIsIt$

whoIsIt = "Joe Roe Doe"

'Check If first, middle, last name are all present
If WordCount(whoIsIt) > 2 Then
   first = Word$(whoIsIt, 1)
   middle = Word$(whoIsIt, 2)
   last = Word$(whoIsIt, 3)

'Check for presence of only first and last name
ElseIf WordCount(whoIsIt) > 1 Then
   first = Word$(whoIsIt, 1)
   middle = ""
   last = Word$(whoIsIt, 2)

'Assume first name only
Else
   first = Word$(whoIsIt, 1)
   middle = ""
   last = ""
End If
```

WordCount( )

See Also Example

The WordCount( ) function returns an integer indicating the number of words are in the specified text. Words are delimited by spaces, tabs, and carriage-return/linefeeds.

Syntax:

WordCount(*text*)

text A string expression containing the text to parse.

Item$( )
ItemCount( )
Line$( )
LineCount( )
Word$( )

Write

See Also Example

After a file has been opened in either output mode or append mode, the Write statement can write information to the file. Each Write statement begins writing at the current location of the file pointer.

Syntax:

Write [#] *fileNum* [, *expr*]...

fileNum A numeric expression, from 0 to 255, that uniquely identifies the open file within your script.

After the *fileNum*, the information to be printed is listed as a sequence of expressions. Expressions to be printed are separated by a comma (,).

How data appears in the file:

- No leading nor trailing spaces are added to numbers.
- Strings are written *with* enclosing quotes.
- A carriage-return/linefeed is written after each Write statement.
- A comma (,) is written to the file after each expression except for the last expression of each line. This is useful when you want to create a file, in which each line is a record that has fields that are separated by commas.

Input #
Input$( )
Line Input #
Open
Print #
Seek

Write # Example

The following writes the first ten positive numbers along with their squares to the open file:

```
Open "testfile" For Output As #1
For i = 1 To 10
   'Each line has the number and its square (for example, 4, 16)
   Write #1, i, i * i
Next i
```

The output resulting from the above statements appears in the file as follows:

```
1,1
2,4
3,9
4,16
5,25
6,36
7,49
8,64
9,81
10,100
```

Notice in the example that the items on each line are separated by commas and that the numbers have neither leading nor trailing spaces.

The following examples write the strings "asdf" and "qwer" to two consecutive lines:

```
Open "testfile" For Output As #1
Write #1, "asdf"
Write #1, "qwer"
```

WriteINI

See Also Example

Windows has initialization files with the file extension .INI that define the Windows environment. To add an entry or change the value of an entry, you can use the WriteINI statement. When WriteINI is called, if the file, section, or entry specified does not exist, it is created.

Syntax:

WriteINI *section*, *entry*, *value*[, *filename*]

section	A string expression containing the name of the section in the .INI file that contains the desired entry. Section names are specified without the enclosing brackets.
entry	A string expression containing the name of the entry whose value is to be changed. If *entry* is an empty string (""), then the entire section is deleted.
value	A string expression containing the new value to assign to the specified entry. If *value* is an empty string (""), the entry is deleted.
filename	A string expression containing the complete or relative pathname for the .INI file to examine. The default file is the WIN.INI file. If no path precedes the name of the .INI file (for example, "CONTROL.INI"), it is assumed to be in the Windows directory. To examine a file not in the Windows directory, include a pathname (for example, ".\MYINI.INI" for the .INI file in the current directory, or "C:\TEST\TEST.INI" for the .INI file in the C:\TEST directory).

Environ$( )
ReadINI$( )
ReadINISection

ReadINI$( ), ReadINISection, and WriteINI Example

The following example reads all the entry names in the "windows" section of the WIN.INI file using the ReadINISection statement, reads the value of the first entry using ReadINI$( ), writes a new value to the first entry using WriteINI, and finally restores the original value.

```
'Declare array to hold entries
Dim entries$( )

'Read the entries from the "windows" section of WIN.INI
ReadINISection "windows", entries

'Make sure at least one entry was found
If ArrayDims(entries) = 1 Then
   'Save the old value of the first entry
   oldValue$ = ReadINI$("windows", entries (LBound(entries)))

   'Give the first entry the value "zero"
   WriteINI "windows", entries(LBound(entries)),"zero"

   'Restore the old value to the first entry
   WriteINI "windows", entries(LBound(entries)),oldValue
End If
```

WS_MAXIMIZED

See Also Example

WS_MAXIMIZED is a numeric constant with a value of 1. The AppGetState( ) function returns this value to indicate that a main window is maximized. In addition, the value is used in the call to AppSetState to maximize a main window.

WS_MINIMIZED
WS_RESTORED

WS_MAXIMIZED, WS_MINIMIZED, and WS_RESTORED Example

The following example determines the state of the active application and displays the result in a message box.

```
'Get the state of the active application
state% = AppGetState( )
If state = WS_MAXIMIZED Then
   MsgBox "Maximized"
ElseIf state = WS_MINIMIZED Then
   MsgBox "Minimized"
ElseIf state = WS_RESTORED Then
   MsgBox "Restored"
End If
```

WS_MINIMIZED

See Also Example

WS_MINIMIZED is a numeric constant with a value of 2. The AppGetState( ) function returns this value to indicate that a main window is minimized. In addition, the value is used in the call to AppSetState to minimize a main window.

WS_MAXIMIZED
WS_RESTORED

WS_RESTORED

See Also Example

WS_RESTORED is a numeric constant with a value of 3. The AppGetState( ) function returns this value to indicate that a main window is restored. In addition, the value is used in the call to AppSetState to restore a main window.

WS_MAXIMIZED
WS_MINIMIZED

XOR Operator

See Also Example

The XOR logical operator yields the logical exclusive OR of two expressions. The result is TRUE if one and only one of the expressions is TRUE. If the expressions are both TRUE or both FALSE, the result is FALSE.

Syntax:

expr1 **XOR** *expr2*

expr1	A numeric, relational, or logical expression.
expr2	A numeric, relational, or logical expression.

If the expressions are numeric, the result is a bitwise XOR of the two expressions. If either of the expressions is a floating-point number, the two expressions are converted to longs before the bitwise XOR.

AND Operator
If...Then...Else...End If
NOT Operator
OR Operator

XOR Operator Example

The XOR operator can be used to test that one and only one condition holds.

```
/* Give free admission
   to anyone named Hercules who is 2 years or older and
   to anyone not named Hercules who is less than 2 years old */
If personName = "Hercules" XOR age < 2 Then
   freeAdmission = TRUE
End If
```

Year( )

See Also Example

The Year( ) function returns a number in the range from 100 to 9999 representing the year from a serial date.

Syntax:

Year(*serialDateTime*)

serialDateTime	Serial date, a number of type double, from which the year is to be extracted.

DateSerial( )
DateValue( )
Day( )
Hour( )
Minute( )
Month( )
Now( )
Second( )
TimeSerial( )
TimeValue( )
Weekday( )

ScriptMaker Editor Functions

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z

A

about

B

backspace
beginning_of_buffer
beginning_of_line
bottom_of_window

C

D

E

F

G, H, I, J, K

L, M

select_to_end
select_to_eol
select_to_top
select_word
select_word_left
select_word_right
split_line
stamp

T
tab_right
tile
to_bottom
to_center
to_top
toggle_backup
toggle_insert
toggle_wordwrap
top_of_window

U, V
undo
unmark_block
uppercase

W, X, Y
Window Control functions
window_down
window_up
word_left
word_right
wrap_para

Z
zoom_window

about
Displays the ScriptMaker Editor version number, copyright notice, and miscellaneous system information.
Default keystroke: Alt+V

backspace
Deletes selected text; otherwise, deletes the character to the left of the cursor. If the cursor is at the beginning of a line, runs the two lines together.

Default keystroke: BkSp or Shift+BkSp

beginning_of_buffer
Moves the cursor to the beginning of the file in the active window.
Default keystroke: Ctrl+Home

beginning_of_line
Moves the cursor to the beginning of the current line.
Default keystroke: Ctrl+PgDn

bottom_of_window
Moves the cursor to the bottom line of the window.
Default keystroke: Ctrl+PgDn

cascade
Resizes and rearranges all open windows in an overlapping pattern. Same as the Cascade command in the Window menu.

Default keystroke: Alt+W, C

change_case
Within the selected block, changes the case of all letters: makes uppercase letters into lowercase, and lowercase into uppercase.

Default keystroke: (none)

close_window
Closes the active window. You can also close a window by double-clicking its Control-menu box. Same as the Close command in the File menu.

Default keystroke: Ctrl+F4, or Alt+hyphen, C

compare
Prompts for two filenames, then compares them line-by-line. Same as the Compare command in the File menu.

Default keystroke: Alt+F, E

copy
Copies the selected text to the Clipboard. If no text is selected, but the Cut/Copy Line If No Text Is Selected option (under Editor Preferences) is checked, this copies the line the cursor is on. Same as the Copy command in the Edit menu (see copy_line).

Default keystroke: Ctrl+Ins, or numpad +

copy_block
Copies the selected text to the Clipboard. Has no effect if no text is selected.
Default keystroke: (none)

copy_line
Copies the current line to the Clipboard.

Default keystroke: (none)

cursor_down
Moves the cursor down one line.

Default keystroke: DownArrow

cursor_left
Moves the cursor one character to the left or, if in the leftmost column, to the end of the line above.
Default keystroke: LeftArrow

Cursor Motion Functional Group

When deciding what kinds of activities to assign to special keys, you may find it useful to think of them according to functional groups. The following all relate to cursor motion:

beginning of buffer	page up
beginning of line	split line
bottom of window	tab right
cursor down	to bottom
cursor left	to center
cursor right	to top
cursor up	top of window
end of buffer	window down
end of line	window up
enter	word left
goto line	word right
page down	

cursor_right
Moves the cursor one character to the right or, if at the end of the line, to the beginning of the next line.

Default keystroke: RightArrow

cursor_up
Moves the cursor to the same character position in the previous line of text. The cursor moves to the end of the line if the previous line does not have a character in the desired position.

Default keystroke: UpArrow

cut
Removes the selected text to the Clipboard. If no text is selected, but the Cut/Copy Line If No Text Is Selected option (under Editor Preferences) is checked, this cuts the line the cursor is on (see cut_line). Same as the Cut command in the Edit menu.

Default keystroke: Shift+Del, or numpad **-**

cut_block
Removes the selected text, placing it in the Clipboard. This variation of the Cut command has no effect if no text is selected.

Default keystroke: (none)

Cut, Copy, and Paste Functional Group

When deciding what kinds of activities to assign to special keys, you may find it useful to think of them according to functional groups. The following all relate to cutting or copying and pasting:

copy	copy block	copy line
cut	cut block	cut line
paste	undo	

cut_line
Removes the current line, placing it in the Clipboard.
Default keystroke: (none)

delete
Deletes the selected text, but does not copy it to the Clipboard. If no text is selected, deletes the character at the cursor. You can recover the text with the Undo command in the Edit menu.

Default keystroke: Del

delete_line
Deletes the current line, but does not copy it to the Clipboard. You can recover the line with the Undo command in the Edit menu.

Default keystroke: Alt+D

delete_to_eol
Deletes all text from the cursor to the end of the current line, but does not copy it to the Clipboard. You can recover the text with the Undo command in the Edit menu.

Default keystroke: Alt+K

delete_word_left
If the cursor is in a word, deletes the text from the cursor to the beginning of the word. If the character to the left of the cursor is a space or tab, deletes the text from the cursor to the previous non-blank character. If the character to the left of the cursor is a delimiter other than a space or tab, deletes that delimiter.

Default keystroke: Ctrl+BkSp

delete_word_right
If the cursor is in a word, deletes the text from the cursor to the end of the word. If the character to the right of the cursor is a space or tab, deletes the text from the cursor to the next non-blank character. If the character to the right of the cursor is a delimiter other than space or tab, deletes that delimiter.

Default keystroke: Ctrl+Del

Deleting Text Functional Group

When deciding what kinds of activities to assign to special keys, you may find it useful to think of them according to functional groups. The following all relate to deleting text:

backspace	delete to eol
cut	delete word left
cut line	delete word right
delete	replace
delete line	undo

document_preferences
Displays the Document Preferences dialog box, where you specify settings for your documents.

Default keystroke: F4

editor_help
Calls the ScriptMaker Editor help system.
Default keystroke: F1

end_of_buffer
Moves the cursor to the end of the file.

Default keystroke: Ctrl+End

end_of_line
Moves the cursor to the end of the current line.

Default keystroke: End

enter
In Insert mode, inserts a carriage return and line feed, then moves the cursor to the beginning of the next line. If Auto Indent is checked in Document Preferences, it positions the cursor below the first non-blank character in the previous line.

In Overwrite mode, moves the cursor to the beginning of the next line.

Default keystroke: Enter

exit
Prompts you to save any modified files, then closes all windows, and exits the editor. Same as the Exit command in the File menu.

Default keystroke: Alt+F4

exit_windows
Prompts you to save any modified files, then ends the current Windows session. The session does not end unless all applications agree to terminate.

Default keystroke: (none)

Files Functional Group

When deciding what kinds of activities to assign to special keys, you may find it useful to think of them according to functional groups. The following all relate to files:

close window	open file	save file
compare	revert	save file close window
find files containing	save all	toggle backup
list files containing	save all exit	
new file	save all exit windows	

find
Displays the Find dialog box to set search criteria and begins searching the file. Same as the Find command in the Search menu.

Default keystroke: Ctrl+S

find_again
Continues a search begun with the Find command. Same as the Find Again command in the Search menu.

Default keystroke: Ctrl+A

find_files_containing
Displays the Find Files Containing dialog box to set search criteria and begins searching for files containing the search string. Choose List Found Files from the Search menu to display a list of all files matching the search criteria. Same as the Find Files Containing command in the Search menu.

Default keystroke: Ctrl+F

goto_line
Prompts for a line number, then moves the cursor to the specified line. If any text is selected, the selection is extended to include the requested line. Same as the Goto Line command in the Search menu.

Default keystroke: Ctrl+G

Lines Functional Group

When deciding what kinds of activities to assign to special keys, you may find it useful to think of them according to functional groups. The following all relate to lines:

beginning of line	goto line
copy line	select line
cursor down	select line down
cursor up	select line up
cut line	select to bol
delete line	select to eol
delete to eol	split line
end of line	to bottom
enter	to top

list_files_containing
Displays a list of files found by the Find Files Containing function. Double-click a filename to open the file. Same as the List Found Files command in the Search menu.

Default keystroke: Ctrl+L

lowercase
Converts all uppercase characters in a selected block to lowercase. If no block is selected, acts on the character at the cursor.

Default keystroke: (none)

new_file
Opens a blank, untitled document window. Same as the New command in the File menu.
Default keystroke: Alt+F, N

new_window
Opens an additional window for the active file. Same as the New Window command in the Window menu.
Default keystroke: Alt+W, N

next_window
Activates the next window in the Editor's Circular list. If you are at the end of the list of windows, next_window selects the first window in the list.

Default keystroke: Ctrl+F6, or Alt+N

open_file
Prompts for a directory and filename and opens a window on that file. Same as the Open command in the File menu.

Default keystroke: F3

page_down
Moves the cursor down one screen page; that is, the number of lines visible in the window.
Default keystroke: PgDn

page_up
Moves the cursor up one screen page; that is, the number of lines visible in the window.
Default keystroke: PgUp

Paragraphs Functional Group

When deciding what kinds of activities to assign to special keys, you may find it useful to think of them according to functional groups. The following all relate to paragraphs:

document preferences	toggle wordwrap
enter	wrap para
split line	

paste
Inserts the contents of the Clipboard at the insertion point. Same as the Paste command in the Edit menu.
Default keystroke: Shift+Ins

play_macro
Replays the keystrokes and functions recorded by the most recent use of the record_macro function. If you have not recorded a macro in this session, the message No Macro Defined appears on the status line. Same as the Play Back Macro command in the Edit menu.

Default keystroke: F8

prev_window
Activates the previous window on the ScriptMaker Editor circular list.
Default keystroke: (none)

print
Prints the file in the active window. Same as the Print command in the File menu.
Default keystroke: Alt+P

record_macro
Starts recording keystrokes and editor functions. Recording continues until the next use of this key. Same as the Record Macro/Stop Recording Macro commands in the Edit menu.

Default keystroke: F7

replace
Prompts for search and replace criteria, then replaces specified text in the file, starting at the cursor location. Same as the Replace command in the Search menu.

Default keystroke: Ctrl+R

restore_window
Resizes the active window to its "intermediate" size (between maximized and minimized). Same as the Restore command in the Control menu.

Default keystroke: Ctrl+F5, or Alt+hyphen, R

revert
Prompts for confirmation, then undoes all changes to the contents of the file since you last saved it. Same as the Revert command in the File menu.

Default keystroke: Alt+F, V

save_all
Saves the contents of all modified files to disk. For untitled files, prompts for filenames. Same as the Save All command in the File menu.

Default keystroke: Alt+F, L

save_all_exit
Saves the contents of all modified files to disk, then ends the ScriptMaker Editor session.
Default keystroke: Alt+X

save_all_exit_windows
Saves the current contents of all modified files to disk, then ends both the ScriptMaker Editor and Windows sessions. The Windows session is not terminated unless all applications agree.

Default keystroke: (none)

save_file
Saves the file in the active window to disk. If the file is untitled, ScriptMaker Editor prompts for a filename. Same as the Save command in the File menu.

Default keystroke: F2

save_file_close_window
Saves the contents of the file to disk, then closes the window. If the window is untitled, the ScriptMaker Editor prompts for a filename.

Default keystroke: (none)

Searching and Replacing Functional Group

When deciding what kinds of activities to assign to special keys, you may find it useful to think of them according to functional groups. The following all relate to searching and replacing:

find	list files containing
find again	replace
find files containing	

select_all
Selects (highlights) the entire contents of the active file. Same as the Select All command in the Edit menu.

Default keystroke: Alt+E, A

select_char_left
Selects the character to the left of the cursor, or cancels the selection if that character is already selected. If the cursor is at the beginning of a line, it moves to the end of the previous line.

Default keystroke: Shift+LeftArrow

select_char_right
Selects the character to the right of the cursor, or cancels the selection if that character is already selected. If the cursor is at the end of a line, it moves to the beginning of the next line.
Default keystroke: Shift+RightArrow

Selecting Text Functional Group

When deciding what kinds of activities to assign to special keys, you may find it useful to think of them according to functional groups. The following all relate to selecting text:

select all	select to bol
select char left	select to end
select char right	select to eol
select line	select to top
select line down	select word
select line up	select word left
select page down	select word right
select page up	unmark block

select_line
Selects the current line.

Default keystroke: (none)

select_line_down
Extends the current selection down one line, or cancels the selection if that line is already selected.

Default keystroke: Shift+DownArrow

select_line_up
Extends the current selection up one line, or cancels the selection if that line is already selected.
Default keystroke: Shift+UpArrow

select_page_down
Extends the current selection down one page and scrolls the window.

Default keystroke: Shift+PgDn

select_page_up
Extends the current selection up one page and scrolls the window.
Default keystroke: Shift+PgUp

select_to_bol
Selects the text from the cursor to the beginning of the line, or cancels the selection if the text is already selected.

Default keystroke: Shift+Home

select_to_end
Selects the text from the cursor to the end of the file.

Default keystroke: Ctrl+Shift+End

select_to_eol
Selects the text from the cursor to the end of the line, or cancels the selection if the text is already selected.

Default keystroke: Shift+End

select_to_top
Selects the text from the cursor to the beginning of the file.

Default keystroke: Ctrl+Shift+Home

select_word
Selects the word the cursor is on.

Default keystroke: Mouse double-click

select_word_left
Extends the current selection to the beginning of the word to the left of the cursor, or cancels the selection if the text is already selected.

Default keystroke: Ctrl+Shift+LeftArrow

select_word_right
Extends the current selection to the beginning of the word to the right of the cursor, or cancels the selection if the text is already selected.

Default keystroke: Ctrl+Shift+RightArrow

split_line
Breaks the line at the cursor without moving the cursor.
Default keystroke: Ctrl+N

stamp
Inserts the current date and time at the insertion point. Same as the Time/Date command in the Edit menu.

Default keystroke: (none)

tab_right
In Insert mode, inserts a tab character at the cursor. If Expand Tabs with Spaces is checked in the Document Preferences dialog box, inserts spaces.
In Overwrite mode, moves the cursor to the next tab position, as set in that same dialog box.

Default keystroke: Tab

tile
Resizes and rearranges all open windows to fit within the editor main window, without overlapping. Same as the Tile command in the Window menu.

Default keystroke: Alt+W, T

to_bottom
Moves the current line to the bottom of the window.
Default keystroke: Ctrl+B

to_center
Moves the current line to the center of the window.

Default keystroke: Ctrl+C

to_top
Moves the current line to the top of the window.

Default keystroke: Ctrl+T

toggle_backup
Toggles the Make Backup Files option in the Editor Preferences dialog box. When the backup option is on and you save a file, the original file is stored with the file extension .BAK.

Default keystroke: (none)

toggle_insert
Toggles between Insert and Overwrite mode. The current mode is displayed on the status line.

Default keystroke: Ins

toggle_wordwrap
Toggles in and out of Word Wrap mode. The current mode is displayed on the status line. Same as the Word Wrap option in the Edit menu.

Default keystroke: Ctrl+W

top_of_window
Moves the cursor to the top visible line of the window.

Default keystroke: Ctrl+PgUp

undo
Reverses the effects of the most recent editing operation. Repeated use will undo up to 300 edit operations, depending on the setting in Editor Preferences. Same as the Undo command in the Edit menu.

Default keystroke: Alt+BkSp, or numpad *

unmark_block
Deselects a block of selected text and returns the cursor to its location before the block was marked.
Default keystroke: Esc

uppercase
Converts all lowercase characters in a selected block to uppercase. If there is no marked block converts the character at the cursor.

Default keystroke: (none)

Windows Functional Group

When deciding what kinds of activities to assign to special keys, you may find it useful to think of them according to functional groups. The following all relate to windows:

arrange icons	restore window
cascade	tile
close window	to bottom
new window	to center
next window	to top
prev window	zoom window

window_down
Moves the cursor one line up while moving the text in the window one line down.
Default keystroke: Ctrl+UpArrow

window_up
Moves the cursor one line down while moving the text in the window one line up.
Default keystroke: Ctrl+DownArrow

word_left
Moves the cursor to the beginning of the previous word.
Default keystroke: Ctrl+LeftArrow

word_right
Moves the cursor to the beginning of the next word.
Default keystroke: Ctrl+RightArrow

wrap_para
Reformats the current paragraph within the margins set in Document Preferences. Same as the Wrap Paragraph command in the Edit menu.

Default keystroke: F12

zoom_window
Resizes the active window to its maximum possible size within the ScriptMaker Editor window. Same as using the Maximize command in the Control menu, or restoring a window by double-clicking its title bar.

Default keystroke: Ctrl+F1 or Alt+hyphen, X

Script Overview

See Also

The script is the basic programming unit. A script is an ASCII text file containing subroutines and functions, each of which performs a particular task. Each subroutine or function has a name and is executed when its name is used in another subroutine or function. For an explanation of the differences between subroutines and functions, see the Subroutine and Function Overview.

Every script has a subroutine named **Main**. **Main** controls the script's execution. It is the first to be executed and it causes other subroutines and functions to be executed by calling their names. **Main** calls the other subroutines and functions or they call each other. **Main** can be the only subroutine in the script, but when there are other subroutines and functions, it is the last one listed in the file.

Main starts with `sub Main`. **Main's** last line ends the script: `end sub`.

NOTE: If you are not writing long or complicated scripts, **Main** is probably the only subroutine in your script; the first line of your script is **Sub Main**.

Statement Overview

See Also

A statement is an executable line of a script. A carriage return/linefeed separates each statement from the one that follows it. [[The following sections explain the parts of a statement and introduce several kinds of statements, including assignment statements and declaration statements.]]

Often a statement is part of a construct, a sequence of statements that has a particular pattern or order. Constructs can control which statements are executed in which order. Constructs include such items as subroutines, functions and loops.

Statement Components

See Also

Statements are composed of reserved words and expressions.

A reserved word is a word that has a special meaning in the programming language and can be used only as its syntax allows. A reserved word cannot be used to name files, variables, and so forth. In this online help, a reserved word used in the syntax is in bold type to remind you to use it exactly as it appears.

An expression consists of one or more operands separated by operators and is evaluated to form a result. To use an example from algebra, *x* + *y* is a numeric expression with two operands (x and y) and one operator (+).

Operands
Operators
Data Types
Variables
Constants

Operands

See Also

An operand used in a statement can be one of the following:

variable A variable is the name of a location in memory that stores a value. The value of a variable usually changes during script execution. The script uses the name of the variable, such as *x*, to represent the value currently stored in *x*'s location. Every variable has a name, a value, and a data type. The *data type* tells what kind of value is stored in the variable. The data type determines how the value can be manipulated. For example, if a variable's data type is integer, its value can be added, subtracted, and so forth.

literal A literal is a value of a particular type, rather than a representation of that value. An example of a *numeric literal* is the number 4. A numeric variable can store the value 4, but its name is not 4, the value itself. A *string literal* is the sequence of characters in the string. For example, "Hello, world." is a string literal. String literals are enclosed in delimiter characters. In ScriptMaker, the only string delimiters used are the double quotation marks.

In some user's guides, a literal is called a constant, because its value does not change. However, this guide uses the term *constant* only in the context of the predefined and user-defined constants explained briefly below and in depth later in More About Constants.

constant A constant is like a variable in that it has a name and represents a value. Unlike the value of a variable, the value of a constant cannot change during the execution of the script. ScriptMaker has both predefined and user-defined constants.

function A function is a named sequence of statements that performs a task. The statements are executed when the name of the function appears in an expression. ScriptMaker has user-defined and predefined functions, both of which are explained in User-defined Functions and Predefined Functions.

Operators

Operators

See Also

Operators indicate what operations, such as addition and subtraction, are to be performed on the operands in an expression. Some operators have different meanings depending on the data type being operated on. For example, the plus sign (+) indicates addition between numbers and concatenation between strings.

In general, an expression's operands and result must all be the same data type, and the operators must be valid for that data type.

The following are all examples of expressions.

```
'Numeric expression (result is number)
x + y
'String expression (result is string)
"Good " + "Day"
'logical expression (result is true or false)
'Abs is a function that finds absolute value
x > Abs(y)-5
```

The following outline of a script contains one user-defined function and two subroutines. The syntax (rules for constructing) subroutines and functions is explained in the Subroutine and Function Overview.

```
Sub One ( )
    ...
End Sub
Function First ( ) As Integer
    ...
    First = ...
    ...
End Function
Sub Main
    ...
End Sub
```

Operands

Data Types

See Also

A data type is a kind of value and determines what operators are valid for that value. For example, if a value's data type is string, the value can be concatenated to other strings.

ScriptMaker supports the following data types:

- integer
- long (long integer)
- single (single-precision floating-point number)
- double (double-precision floating-point number)
- string
- array

Integers, longs, singles, doubles, and strings are simple data types, types that contain only one value. The table below provides details about them. An array is a composite data type. It can have multiple parts, each of which has a value. Each part of a composite data type is called an array element. An array consists of a number of elements of the same simple type.

Simple Type	Type Declarator	Significant Digits	Size	Range
Integer	%	4	2 bytes (16 bits)	-32768 to 32767
Long	&	9	4 bytes (32 bits)	-2147483648 to 2147483647
Single	!	7	4 bytes (32 bits: 1 for sign, 8 for the exponent, and 23 for the mantissa)	approximately +/-3.4E+/-38
Double	#	15-16	8 bytes (64 bits: 1 for sign, 11 for the exponent, and 52 for the mantissa)	approximately +/-1.7E+/-308
String	$	N/A	1 byte per character	0 to 32768 characters

Operands
Operators
Variables
Constants

Variables

See Also

A variable is the name of a location in memory that stores a value. The value of a variable can change during script execution. Every variable has a name, a data type, and a value.

To declare a variable is to provide its name and data type. ScriptMaker assumes the first appearance of a variable's name in a subroutine or function is its declaration. If that first use does not explicitly reveal the variable's data type, the compiler implicitly decides on a type.

The explicit declaration of a variable uses a type declarator or a **Dim** statement or both. For an implicit declaration, the first letter of the variable's name determines its data type. Any misspelling of a variable's name can become an implicit declaration of another variable.

ScriptMaker gives each variable an initial value at the time it is declared. A string variable is initialized to the empty string, a string with no characters (""). A numeric variable is initialized to zero. ScriptMaker has only local variables. In general, a local variable is a variable that is only known to and used by the subroutine or function where it is declared.

Explicitly Declaring Variables
Implicitly Declaring Variables
Operands
Operators
Data Types
Constants

Explicitly Declaring Variables

When the variable is a simple type (integer, long, single, double, or string), use a symbol called a *type declarator* after the variable's name the first time it appears in the script. It does not have to appear in a particular kind of statement. The type declarator for an integer is %, for a long is &, for a single is !, for a double is #, and for a string is $. The compiler recognizes the first use of the variable as an indication of its existence and type. Subsequent uses of that variable do not need the type declarator.

For example, the following statements tell the compiler that the variables `Number_of_guests`, `Number_of_members`, and `Total_number` are of type long.

```
Number_of_guests& = 45
Number_of_members& = 100
Total_number& = Number_of_guests + Number_of_members
```

For any type of variable, you can use a **Dim** statement.

Syntax:

Dim *VarName* [**As** *type*] [**,** *VarName* [**As** *type*]]...

If you use a type declarator at the end of the variable's name, the `as type` clause is unnecessary, and vice versa. You can use both so long as they indicate the same type. All **Dim** statements appear inside user-defined functions or subroutines. You must use a **Dim** statement to declare an array. See Dim for details about array declarations.

Both of the **Dim** statements in the following example declare a string variable.

```
Sub Main
Dim first_name As String  'User's first name
Dim last_name$ 'User's last name
...
End Sub
```

The **Dim** statement in the next example declares more than one variable.

```
Sub Main ()
Dim Total_number&, Number_of_guests&
...
End Sub
```

Using a separate **Dim** statement for each variable, along with an explanation of the variable's purpose at the beginning of each subroutine and function, makes the script easier to debug and maintain.

Implicitly Declaring Variables

You can use a **Def** statement to specify a simple type and the initial letters for variables of that type.

Syntax:

Def*type letters*

where *type* is replaced by **Int** for integer, **Lng** for long, **Dbl** for double, **Sng** for single, or **Str** for string, and *letters* is replaced by a series of letters of the alphabet separated by commas. A range of letters can be specified by placing a hyphen between the first and last letters of the range. The syntax for *letters* is:

letter [**-** *letter*] [**,***letter* [**-** *letter*]]...

The **Def** statements in the following example make any variables that are not explicitly declared into integers if their names start with I, M, or Q; into longs if their names start with A, B, C, or N; and into strings if their names start with T through Z.

```
DefInt I, M, Q
DefLng A-C, N
DefStr T-Z
Sub Main
...
End Sub
```

Def statements must appear outside of user-defined functions and subroutines, not within them. This makes them global type definitions that are valid for any subroutine or function that follows them. (It does not make the variables whose types are defined by the **Def** statement global variables.) Additional **Def** statements cannot contradict earlier ones. For example, you cannot define A-F as integers and later define C as a string. To use the same **Def** statements throughout the script, make them the first statements in the script.

NOTE: If a variable does not appear in a **Dim** statement, nor end in a type declarator, nor start with a letter listed in a **Def** statement, the compiler assumes it is an integer. Because of these implicit declarations, misspellings can result in new variables that you never intended. You may want to check your variable names if a script compiles successfully but does not run correctly.

Constants

See Also

A constant is like a variable in that it has a name and represents a value. Unlike the value of a variable, the value of a constant cannot change during the execution of the script. ScriptMaker has both predefined and user-defined constants.

User-defined Constants
Predefined Constants

User-defined Constants

User-defined constants are constants that you create outside of functions and subroutines. This makes them global constant declarations recognized by all the user-defined functions and subroutines that follow them. Each constant is valid only in the script in which it appears.

If you use some string or numeric literal repeatedly, such as "Invalid input." or 459, define it as a user-defined constant. Using constants:

- Makes the code more readable. For example, 459 could become the constant Number_Of_Users.
- Saves memory because the literal is not repeated and, therefore, not stored in more than one place in memory.
- Allows you to change the literal by changing just one line rather than several lines throughout the script. For example, you can change the message "Invalid input." to the more user-friendly message "You must enter a number." by changing only the line where the message is declared as a constant.

Syntax:

Const *constantName* **=** *expression* [**,** *constantName* **=** *expression*]...

The expression can use only string or numeric literals, the predefined constants TRUE or FALSE, or previously declared user-defined constants. Functions are not allowed. You do not have to use type declarators.

```
Const Message1 = "Are you sure?", Message2 = "Please wait..."
Sub Main
MsgBox Message2
...
End Sub
```

Predefined Constants

Predefined constants are reserved words in the language. They represent values needed in certain statements. Online help explains each predefined constant as part of the statement that uses it. Each predefined constant has a numeric value. For example, ATTR_ARCHIVE, used in the FileList statement, has the value 32. However, you should use the constant name in your scripts for readability and maintainability. The numeric values for predefined constants can change from version to version, but the constants' names do not.

Assignment Statements

See Also

The assignment statement is one of the most-often-used statements. It assigns the value of the expression on the right side of the assignment operator (=) to the variable or element of an array on the left side of the operator. The assignment statement must do the following:

- Identify the variable or array element that receives the value.
- Use the assignment operator (=) to separate the variable or element from the expression.
- End with the expression that determines the variable's value.

Optionally, it can start with the reserved word **Let**. This word is left over from the earliest versions of BASIC.

NOTE: The assignment operator is the equal sign (=). The equal sign is also used as a relational operator that compares two quantities to see if they are equal. The difference is that the assignment operator gives a variable a value, and a comparison for equality returns a value of true (if equal) or false (if unequal).

The initial value of a numeric variable is zero. A string variable has the empty string as its initial value. An assignment statement changes that value.

The syntax for assigning a value to a variable is:

[**Let**] *varName* **=** *expression*

Both of the following examples assign the value 5 to x.

```
Let x = 5
x = 5
```

You can use a variable on both sides of the first assignment statement that uses it. For example, the following statement increases the value of the variable `Counter` by one.

```
Counter = Counter + 1
```

When this statement is executed, the value of the `Counter` on the right side is 0, its initial value, and the value of the `Counter` on the left is the sum of 0 + 1, which is 1.

Constructs

A construct is a sequence of statements that follow a pattern and serve a purpose within the script. Failing to follow the pattern causes compiler errors. For example, control constructs control which statements in a script are executed and which statements are not. They can choose a group of statements to execute from several such groups, repetitively execute a group of statements, and transfer control from one part of the script to another. The control constructs are conditional constructs (such as **If** statements and **Select Case** statements), loops (such as **For**, **While**, and **Do** loops), goto statements (such as **GoSub** and **Goto**), and the **End**, **Stop**, and **Sleep** statements.

Subroutine and Function Overview

See Also

Subroutines and functions are very similar. Each is a sequence of statements that performs a task. Each has to be declared or defined, and each is executed when its name is used in another subroutine or function. Each can change the values of variables that are passed to it by reference. Passing parameters by reference is explained in Parameters in Calls.

The differences between subroutines and functions are:

- A subroutine's name never returns a value to the subroutine or function that calls it, and a function's name always does.
- The way a subroutine is called differs from the way a function is called. A subroutine's name appears in a **Call** statement. A function's name is part of an expression.

In the sections that follow, a subroutine named **Square** and a function named **Square** both perform the same task. This allows you to see the differences between a subroutine and a function more clearly.

Calling a Function or Subroutine

Subroutines

See Also

The parts of a subroutine declaration are:

- The statement that identifies it as a subroutine, tells the subroutine's name, and identifies its parameters.
- Executable statements.
- The statement that ends the subroutine declaration.

Syntax for subroutine:

Sub *subName* [**(** [*parameterList*] **)**]
    [*statements*]
End Sub

The syntax for calling a subroutine:

[**Call**] *subName* [**(** [*parameterList*] **)**]

Parameters are passed by reference unless explicitly passed by value. See Parameters for details about parameters and parameter passing.

The following example shows the declarations or definitions of the subroutines named **Square** and **Main**. **Main** is the first subroutine to be executed. The **Call** statement in **Main** calls the **Square** subroutine. **Square** squares the value of the variable *sum* that is passed to it as the parameter *x*. Since *sum* is passed by reference, changes made to its value by **Square** are known to **Main** as well. In this example, *sum* has the value 7 before the call to **Square** and the value 49 after the call.

```
'declaration of Square subroutine
Sub Square (x&)
    'The variable sum becomes known to Square as x
    x = x * x
End Sub
'declaration of Main subroutine
Sub Main ()
    ...
    x = 3
    y = 4
    'sum equals 7 here
    sum = x + y
    'Execution of Square occurs
    Call Square (sum)
    ...   'sum equals 49 here
End Sub
```

Predefined Subroutines

See Also

A number of statements are really predefined subroutines. For example the FileList statement can be executed in either of the following forms:

```
FileList files, "c:\*.bat"
```

Or,

```
Call FileList (files, "c:\*.bat")
```

User-defined Functions

See Also

You create a user-defined function to perform a task and return a value. Usually you create a function for a task that the script needs to perform more than once.

A function declaration consists of:

- The statement that identifies it as a function, tells its name, identifies its parameters, and provides a simple type for the function's name as though it were a variable.
- Executable statements, one of which assigns a value of the correct type to the function's name. This value is returned by the user-defined function to the statement using the function.
- The statement that ends the function declaration.

Syntax:

Function *functionName* [**(** [*parameterList*] **)**] [**As** *type*]
    [*localDeclarations*]
    [*Statements*]
End Function

Each function has a simple type: string, integer, long, single, or double. Parameters are passed by reference unless explicitly passed by value. See Parameters for details about parameters and parameter passing.

The following example shows the declarations of the **Square** function and the **Main** subroutine. Each use of the function's name (**Square**) inside **Main** calls the function. A statement in **Main** uses **Square** twice in the same expression. **Square** is used as though it were a variable of type long because the function is type long, and the value assigned to **Square** inside the **Square** function is used to evaluate the expression inside **Main**. **Square** squares *a* (which is passed to it as a parameter the first time) and returns the value 9 (which is 32) to the statement. Then **Square** squares *b* (which is passed to it the second time) and returns the value 16 (which is 42) to the statement. The statement assigns the value 25 (9 + 16) to *c*.

```
'declaration of Square function
Function Square (x&) As Long
    'x takes the value of the a, then b
    Square = x*x
End Function
'declaration of Main subroutine
Sub Main ()
    ...
    a = 3
    b = 4
    'calls Square twice
    c = Square(a) + Square(b)
    ...
End Sub
```

User-defined Functions and Subroutines

Predefined Functions

See Also

ScriptMaker has a large library of commonly used predefined functions.

A predefined function saves you time because you don't have to write it. To use one, you only need to know its purpose, syntax, and the type of result that it returns to your script. You use the function as though it were its result. A function is not a statement and never appears alone on a line of your script. Most often, you use it as part or all of an expression in an assignment statement.

In this example, the **Len** function counts the characters in a string and returns that length to the script. This saves you the time it would take to write statements that count the characters in a string. Its syntax is **Len(***exprS***)**, where *exprS* is any string expression. To find the length of a string variable named VendorName, you would use something like:

```
Length = Len(VendorName)
```

If `VendorName` is "Ajax Corporation", the function would return the number 16 (15 letters and 1 space).

Calling a Function or Subroutine

Case Sensitivity

See Also

ScriptMaker is not case-sensitive. SUB MAIN is equivalent to Sub Main and sub main. You can type everything in one case or use capitalization to increase readability.

Comments
Identifiers
Scope
User Interface
Programming Environment

Comments

See Also

Comments in a script file are explanations of what the script does. Because they are set off by special characters, the compiler ignores them. Good comments save debugging and maintenance time by explaining:

- The purpose of each script at the beginning of the script.
- The purpose and parameters for each subroutine and user-defined function before the subroutine or function starts.
- Every variable as it is introduced.
- The beginning and ending of each construct.

To comment a whole line or partial line:

- Start the line with **Rem** followed by a space.

```
REM This script performs...
```

Or,

- Start the comment with a single quotation mark (').

```
MsgBox "Hello, world!" 'Displays a string inside a message box
' This script performs...
```

The compiler ignores all characters between the single quotation mark or **Rem** and the end of the line.

To comment more than one line:

- Start the comment with `/*` and end it with `*/` as in the C programming language.

```
MsgBox "Hello, world!" /* This displays a string inside a message
box. The script pauses until the user clicks the OK button. */
```

No statements can appear on the same line as the ending comment marker. The `*/` can be followed only by spaces and the carriage return.

Case Sensitivity
Identifiers
Scope
User Interface
Programming Environment

Identifiers

See Also

The names of variables, constants, user-defined functions, subroutines, and so forth, are called identifiers. An identifier starts with a letter of the alphabet, but subsequent characters can be alphabetic, the digits 0 to 9, or the underscore character (_). A variable identifier can have as many as 255 characters. Creating meaningful identifiers makes your scripts easier to read. Using *x* and *y* as identifiers may be useful in a numeric expression, but they are rarely useful elsewhere. You cannot use characters (such as the period) or reserved words (such as **Main**) as identifiers.

No identifier can be duplicated within the scope of its owner. See Scope for more details. For example, a subroutine cannot have a variable name that is the same as the name of a user-defined function defined before the subroutine in the script. However, two subroutines can have the same identifier for a local variable, even if the variables are not the same type.

The following example uses an identifier for each of three variables.

```
Total_number = Number_of_guests + Number_of_members
```

Case Sensitivity
Comments
Scope
User Interface
Programming Environment

Scope

See Also

A ScriptMaker script uses static scoping and has no forward declarations. his means that functions and subroutines can call themselves and the components of the script declared prior to them within the scriptbut cannot call a component that comes after them. The variables declared inside a subroutine or function are local. This means that they are used only by the subroutine or function in which they appear. A subroutine or a function never uses another's variables. However, the values of those variables or their memory addresses can be passed to the subroutine or function as parameters.

The following example of a script shows where to put executable statements and declarations of various types.

```
'Def statements for entire script
'constant declarations for entire script
Sub One ( )
    'local variable declarations
    'executable statements
End Sub
Function First
    'local variable declarations
    'executable statements
    'assignment of a value to First
End Function
'constant declarations for Main subroutine
Sub Main ( )
    'local variable declarations
    'executable statements
End Sub
```

Main can call itself, **One** and **First** because the definitions of **One** and **First** precede **Main**. **First** can call itself and **One**, but it cannot call **Main**. **One** can only call itself.

The constants declared before **One** apply to the entire script; those declared just before **Main** only apply to **Main** because only **Main** follows those declarations.

The local variables in **One** cannot be used by **First** or **Main** because no subroutine or function can see another's local variables.

Case Sensitivity
Comments
Identifiers
User Interface
Programming Environment

User Interface

See Also

In Windows (and other graphical user interfaces), a dialog box is a special window displayed by ScriptMaker or some other application to communicate with a user. A dialog box displays messages for and requests data from a user. When that data is used to determine what statements to execute, the script is said to be event-driven.

ScriptMaker has several simple predefined dialog box templates. You can display them from any script by using the statements and functions provided for them. If you don't do a lot of programming in ScriptMaker, the predefined dialog boxes may be all you need.

ScriptMaker also provides a Dialog Editor, a tool with which you can create your own templates for dialog boxes. Each dialog box template defines a dialog box's size, its components (such as push buttons and text boxes), the size and position of those components, and so forth. User-defined templates can be included in any ScriptMaker script and give you control over the look and feel of the dialog box and the amount of data that can be obtained from it. However, scripts that use them are more complicated than those that use the predefined dialog box templates.

Case Sensitivity
Comments
Identifiers
Scope
Programming Environment

Programming Environment

See Also

The Editor provides the text editing and script-testing environment that you need to write, debug, and maintain your scripts. It has commands that compile, run, and abort scripts. It allows you to create custom dialog boxes using the Dialog Editor, and it provides a macro recorder and an online reference of all the statements, predefined functions, operators, and so forth in the language.

The Editor has the commands for using files, editing text, and searching for text that you would expect from a text editor. For example, you can save text files with the .SM extension. However, you can also save the compiled code (with the extension .SMC) or convert the script to an .EXE.

The Editor allows you to reassign keystrokes, autosave the current file, and set a number of preferences.

Menu Commands

Expand

File Menu

Edit Menu

Search Menu

Script Menu

Tools Menu

Options Menu

Window Menu

Menu Commands

File Menu

New

Open...

Close

Save

Save As...

Save All

Insert...

Revert

Write Block...

Printer Setup...

Page Setup...

Print

Mail Document...

Compare...

Exit

(list of files)

Edit Menu

Search Menu

Script Menu

Tools Menu

Options Menu

Window Menu

Menu Commands

File Menu

Edit Menu

Undo

Cut

Copy

Paste

Delete

Select All

Time/Date

Word Wrap

Wrap Paragraph

Record Macro

Playback Macro

Search Menu

Script Menu

Tools Menu

Options Menu

Window Menu

Menu Commands

File Menu

Edit Menu

Search Menu

Find...

Find Again

Replace...

Find Files Containing...

List Found Files

Goto Line...

Script Menu

Tools Menu

Options Menu

Window Menu

Menu Commands

File Menu

Edit Menu

Search Menu

Script Menu

Compile

Run

Abort

Save Code

Save EXE

Tools Menu

Options Menu

Window Menu

Menu Commands

Menu Commands

File Menu

Edit Menu

Search Menu

Script Menu

Tools Menu

Options Menu

Customize...

Toolbar

Status Bar

Document Preferences...

Window Menu

Menu Commands

Menu Commands

Collapse

File Menu

- New
- Open...
- Close
- Save
- Save As...
- Save All
- Insert...
- Revert
- Write Block...
- Printer Setup...
- Page Setup...
- Print
- Mail Document...
- Compare...
- Exit
- (list of files)

Edit Menu

- Undo
- Cut
- Copy
- Paste

 Delete

 Select All

 Time/Date

 Word Wrap

 Wrap Paragraph

 Record Macro

 Playback Macro

 **Search Menu**

 Find...

 Find Again

 Replace...

 Find Files Containing...

 List Found Files

 Goto Line...

 **Script Menu**

 Compile

 Run

 Abort

 Save Code

 Save EXE

 **Tools Menu**

 Recorder

 Dialog Editor

 Reference

 **Options Menu**

 Customize...

 Toolbar

 Status Bar

 Document Preferences...

 **Window Menu**

 New Window

 Cascade

 Tile

 Arrange Icons

 1 to n

New command (File menu)

Use this command to open a new untitled document. When you are ready to save the file, use the Save or Save As commands and assign a filename to it.

Open command (File menu)

Use this command to select a file to edit. When the Open dialog box appears, you can type the filename you want into the File Name text box, or use the Files, Drives, List Files of Type, and Directories list boxes to locate a file.

Close command (File menu)

Use this command to close the file in the active window. If you have changed the file in any way since the last time you saved it, the ScriptMaker Editor prompts you to save your changes before closing the file.

Save command (File menu)

See Also

Use this command to immediately save the open file in the active window.

Saving a File with a New Name
Saving Text as a Separate File

Save As command (File menu)

See Also

Use this command to save a file under a new name, to a new path, if desired. When the Save As dialog box appears, type the filename into the File Name text box, or use the Files, Drives, List Files of Type, and Directories list boxes to select a filename and location.
When you use this command, you create a new copy of the file with a new filename. The older version of the file still exists under the old name. If you simply want to rename the file, delete the earlier filename to avoid confusion.

CAUTION: If you choose the name of a file that already exists, ScriptMaker overwrites the older version of the file.

Saving a File with a New Name

Save All command (File menu)

Use this command to automatically save every open file. If you have made changes to any of the open files, the ScriptMaker Editor prompts you to save the changes.

Insert command (File menu)

See Also

Use this command to copy an existing file into the file you are working on. The ScriptMaker Editor inserts the entire file at the current cursor position.

Inserting One File into Another

Revert command (File menu)

Use this command to undo all changes you have made to the file since the last time you saved it. The file *reverts* to its most recent, saved condition.

Write Block command (File menu)

See Also

Use this command to highlight a block of text and save it as a new, separate file.

Saving Text as a Separate File

Printer Setup command (File menu)

See Also

Use this command to select the printer you are going to use. Further options allow you to specify paper source, size, orientation and number of copies to print. You can also choose to print to a file, set margins, set a header, and select font types.

Printing All or Part of a File

Page Setup command (File menu)

See Also

Use this command to set default headers and footers, establish margins, and select fonts.

Printing All or Part of a File

Print command (File menu)

See Also

Use this command to send the file in the active window directly to the printer.

Printing All or Part of a File

Mail Document command (File menu)

See Also

Use this command to send files to other users if you have a mail application that supports the MAPI standard (such as Microsoft Mail) or the VIM standard (such as cc:Mail for Windows or Lotus Notes). For details about mailing documents, see the main Norton Desktop help.

Compare command (File menu)

See Also

Use this command to compare the contents of two document files on a line-by-line basis. The ScriptMaker Editor shows you where the documents differ and where they match. If you are not sure of the documents' exact filenames, click Browse to search for the ones you want.

The comparison starts either at line 1 or at a specific line in the file. You can choose to display the files either one above the other (Horizontal) or side-by-side (Vertical). The ScriptMaker Editor shows you the contents of the files as it compares them, and you can stop the comparison at any point.

Comparing Two Files

Exit command (File menu)

Use this command to quit the ScriptMaker Editor. If you have made changes to any files since the last time you saved them, the ScriptMaker Editor asks whether you want to save the changes before exiting.

List of Files (File menu)

The ScriptMaker Editor remembers the four files you opened most recently, whether or not they are all currently open. Each of these files is listed at the bottom of the File menu. Choosing any item in the list opens that file, making it active. If the file is already open, the ScriptMaker Editor brings that window to the front.

Undo command (Edit menu)

See Also

Use this command to undo, or reverse, keystrokes. For example, if you delete a line and then change your mind, choose Undo to restore the line. You can also use Undo to delete lines you have just entered. Use the Editor Preferences dialog box to specify the number of keystrokes (up to 300) that you can undo at one time.

NOTE: You can never undo operations further back than the last Save.

Setting Editor Preferences

Cut command (Edit menu)

See Also

Use this command to delete a block of text and temporarily store it in the Windows Clipboard so you can move it to another point in the file. If you use the Cut command again without pasting, it deletes the text entirely.

If you do not select (highlight) any particular text, the command cuts whatever line the cursor is on at the moment. Move the insertion point and use the Paste command to move the text.

Moving and Copying Text

Copy command (Edit menu)

See Also

Use this command to copy text to the Windows Clipboard so you can paste it elsewhere. You can copy text within the same file or from one file to another.

If you do not select (highlight) any particular text, the command copies whatever line the cursor is on at the moment. Move the insertion point and use the Paste command to copy the text.

Moving and Copying Text

Paste command (Edit menu)

See Also

This command is always used in conjunction with the Copy or Cut commands. When you move the insertion point and choose Paste, the text you moved or copied to the Clipboard appears in the new position. If there is nothing in the Clipboard, the Paste command is not available.

Moving and Copying Text

Delete command (Edit menu)

Use this command to immediately delete selected text from the file without saving it to the Clipboard. If you change your mind about the deletion, you can reverse it with the Undo command. Delete is available only when you have text selected.

Select All command (Edit menu)

Use this command to select the entire contents of a file. This can be useful when you want to make some global change, such as changing the font for the entire file.

Time/Date command (Edit menu)

Use this command to insert the current date and time at the insertion point. Use the Windows Control Panel to specify your date and time format.

Word Wrap command (Edit menu)

See Also

Use this command to toggle the editor's automatic word-wrapping feature on and off. All typing you do after turning on word wrap uses the current right margin setting to wrap lines of text.

A check mark appears next to the Word Wrap command when automatic word-wrapping is on. Using this command is the same as using the Wrap Text As It Is Typed check box in the Document Preferences dialog box.

Setting Document Preferences

Wrap Paragraph command (Edit menu)

See Also

Use this command to adjust a paragraph to your preset margins. This is useful if inserting text pushes one or more lines past the right margin, or if deleting text leaves a gap in the paragraph.

NOTE: Wrap Paragraph has no effect unless you have also checked Word Wrap on the Edit menu or Wrap Text As It Is Typed in the Document Preferences dialog box.

Setting Document Preferences

Record Macro command (Edit menu)

See Also

This command provides a simple macro utility, which lets you automate a particular editing task by recording a series of keystrokes and playing them back. You can store only one macro at a time, and it is deleted when you exit from the ScriptMaker Editor or when you record another macro. To run the macro, use the Playback Macro command.

Using Editor Macros

Play Back Macro command (Edit menu)

See Also

Use this command to run the macro you recorded with the Record Macro command.

Using Editor Macros

Find command (Search menu)

See Also

Use this command to search for a character string in your current file. The ScriptMaker Editor begins searching at the current cursor position and stops when it finds the string or reaches the end of the file. You can search both forward and backward from the cursor position. To find the next occurrence of the character string, use the Find Again command.

NOTE: Since the ScriptMaker Editor searches line by line, it will not find a character string that breaks onto more than one line.

The Regular Expression option lets you perform nonliteral searches by embedding special characters in your character string. Some of these characters act as wildcards. Others let you search for special characters, such as tabs.

Find Again command (Search menu)

See Also

Use this command to find the next occurrence of the character string you specified with the Find command. Find Again is available only when you have started a search with Find.

Searching for Text in a File

Replace command (Search menu)

See Also

Use this command to replace one character string with another in some or all of the places it occurs in your file. You specify both the text pattern to search for and the pattern you want to replace it with. The ScriptMaker Editor starts from the cursor position and works forward through the file. By default, the ScriptMaker Editor asks whether you want to replace each string as it finds it.

With Replace, you can:

- Use regular expressions.
- Make global replacements (without confirming each change).
- Make your search case-sensitive.

Find Files Containing command (Search menu)

See Also

Use this command to look for a character string in more than one file at the same time. Specify the files you want to search, or use a wildcard to specify a group of files (such as *.TXT). The files do not have to be open.

If you are not sure of the filenames, click Directory on the Find Files Containing dialog box to select the proper directory and files.

When files are found, select a file from the Found Files list to edit or review. To see the list of files again, use List Found Files.

Searching for Text in Multiple Files

List Found Files command (Search menu)

See Also

Use this command to review the list of files that appears after using the Find Files Containing command. If you open one of the files, its name disappears from the list. If only one file is found, the List Found Files command becomes unavailable after you open that file.

Searching for Text in Multiple Files

Goto Line command (Search menu)

Use this command to jump to any line in the current file, if you know the line number. Line numbers are displayed in the status bar at the bottom of the ScriptMaker Editor window.

NOTE: If you highlight text, and choose Goto Line, the highlight extends to include the line number you requested.

Compile command (Script menu)

Use this command to compile the file in the active window and check for syntax errors.

If your script compiles without errors, you can run it from within ScriptMaker or from a command line by entering SMW.EXE *scriptName*, where *scriptName* is the name of your script file. Use SMD.EXE *scriptName* with DOS script files.

If ScriptMaker finds syntax errors, it displays a message box indicating the type of error and the line that the error appears on. Click OK to move the cursor directly to the line containing the error.

Run command (Script menu)

Use this command to execute the file in the active window. If you have a relatively short script, you may want to skip the compilation and just run your script. In this case, ScriptMaker checks for any modifications to the file since the last time it was compiled. If so, ScriptMaker compiles the script and notifies you if there are any problems with the compilation.

Abort command (Script menu)

Use this command to halt script execution. To restart, choose Run from the Script menu.

Save Code command (Script menu)

See Also

Use this command to save the file in the active window as a .SMC file. This is a compiled ScriptMaker file, which can be launched from the ScriptMaker Editor, or from the command line.

Save Exe command

Save EXE command (Script menu)

See Also

Use this command to save your compiled batch file to an executable file that can be run from the command line. This allows you to distribute a ScriptMaker script file that runs independently of the ScriptMaker environment.

NOTE: Use this command only with Windows scripts.

Save Code command

Recorder command (Tools menu)

Use the Macro Recorder to record a series of events that the user generates within the Windows environment, and to translate the recorded series into ScriptMaker statements. The statements can be included in a program or subroutine that can reproduce the series of recorded events.

Dialog Editor command (Tools menu)

Use the Dialog Editor to create your own dialog boxes that can be incorporated into ScriptMaker programs. You can start with an empty Dialog Editor window and build a custom dialog box piece by piece; or you can "capture" an existing dialog box from another Windows application into the Dialog Editor main window and modify that dialog box to suit your purposes.

Reference command (Tools menu)

Use the Reference dialog box as a handy online method of quickly determining the syntax of each ScriptMaker command and function. The Reference dialog box also provides a short description of the command and an example of its use.

Customize command (Options menu)

See Also

Use this command to display the Customize dialog box, which lets you specify preferences for the ScriptMaker Editor, shortcut key assignments, and toolbar.

Toolbar command (Options menu)

Use this command to toggle the toolbar on and off. The status bar provides another method of choosing some of the most common menu commands, using your mouse.

A check mark appears next to the Toolbar command when the toolbar is displayed.

Status Bar command (Options menu)

Use this command to toggle the status bar on and off. The status bar provides information such as the current line and column and whether the file has been changed since it was last saved. In addition, the result of operations, such as a search, appear here.

A check mark appears next to the Status Bar command when the status bar is displayed.

Document Preferences command (Options menu)

See Also

Use this command to open the Document Preferences dialog box so that you can review or change the editor's current settings.

Setting Document Preferences

New Window command (Window menu)

This command lets you open more than one window for a file. When you open multiple windows, you can arrange them one above the other or side by side to view different parts of the file at the same time.

Cascade command (Window menu)

The Cascade command arranges all your open file windows in an overlapping pattern so that only the title bars are visible. Click anywhere on a cascaded window to make it active. To fill the screen with a particular display, make it active, and click the maximize button.

Tile command (Window menu)

The Tile command arranges all your open file windows in a nonoverlapping pattern, which allows you to see part of each one. You can then iconize any window to reduce clutter and maximize the ones you want to view or work with. Click anywhere on a tiled window to make it active. Click the maximize button to give it the full-screen display.

Arrange Icons command (Window menu)

If you have iconized some of the files you are working on, use the Arrange Icons command to arrange the icons evenly along the bottom of the main window. You can then choose to maximize only the ones you want to look at or work with.

1 to n command (Window menu)

When you click the Windows menu, a list of all open windows appears at the bottom of the menu. The windows are numbered for reference. Choosing any numbered item brings that window to the front, making it active.

Open dialog box

Use this dialog box to open a file. If you are not sure of the name, use the Files, Directories, Drives, and List Files of Type list boxes to find the file you want.

File Name text box

Files list box

Directories list box

List Files of Type drop-down list box

Drives drop-down list box

File Name text box
Enter a filename or use wildcards for a range of files. To choose from a specific file type, use the list in the List Files of Type drop-down list box.

Files list box
Lists the names of files in the current directory in alphabetical order.

Directories list box
Displays the directories on a particular drive. (Use the Drives drop-down list box to select the drive.) Double-click the name of a directory to select it. The files in the directory appear in the Files list box.

List Files of Type drop-down list box
Lets you quickly select a specific file type.

Extension	File Type
.TXT	Text files
.INI	Initialization files
.BAT	Batch files
.	All file types in the directory

Drives drop-down list box
Lets you choose a particular drive. Click the prompt button to drop the list of available drives, and click the name of the drive you want.

Close

Save As dialog box

Use this dialog box to save a file under a new name, or to replace the contents of an old file. Enter a filename in the File Name text box, or use the Files, Directories, Drives, and List Files of Type list boxes to find the file you want.

File Name text box

Files list box

Directories list box

List Files of Type drop-down list box

Drives drop-down list box

Close

Insert File dialog box

Use this text box to insert one file into another. Enter a filename in the File Name text box, or use the Files, Directories, Drives, and List Files of Type list boxes to find the file you want.

File Name text box

Files list box

Directories list box

List Files of Type drop-down list box

Drives drop-down list box

Close

Write Block dialog box

Use this dialog box to save part of a file as a new, separate file. Enter a filename in the File Name text box, or use the Files, Directories, Drives, and List Files of Type list boxes to find the file and directory you want.

File Name text box

Files list box

Directories list box

List Files of Type drop-down list box

Drives drop-down list box

Printer Setup dialog box

Use this dialog box to select your printer and set various options.

Printer list box
Shows the name of the printer attached to your computer.

Setup command button
Use this button to set up your printer. You can define:

- Paper source
- Paper Size
- Orientation
- Copies
- Options command button
- Advanced command button

Paper source
Choose from Upper tray, Lower tray, Manual feed, or Envelope feed.

Paper Size
Choose from Letter, Legal, Executive, A4, or three sizes of envelopes.

Orientation
Choose from Portrait or Landscape.

Copies
Enter a number from 0 through 999.

Options command button
Set additional options including Print to file, Margins, and Duplex printing and Scaling; or press the Advanced command button for more options.

Advanced command button
Set further options, including Fonts, Memory, and Graphics functions.

Page Setup dialog box

Use this dialog box to set margins; choose a printer font; and put text, date, and page numbers in a header and footer.

Header drop-down combination box

Footer drop-down combination box

Margins group box

- Left
- Right
- Top
- Bottom

Font command button

Header drop-down combination box
Lets you enter a line of text to be centered at the top of each page. Click the prompt button for a list of any headers that you have recently used. You can include the following codes in the header:

%f	Full path and filename of the document
%d	Current date and time
%p	Page number

NOTE: The header appears only on the printed copy of the file.

Footer drop-down combination box
Lets you enter a line of text to be centered at the bottom of each page. Click the prompt button for a list of footers that you have recently used. You can include the following codes in the footer:

%f	Full path and filename of the document
%d	Current date and time
%p	Page number

NOTE: The footer appears only on the printed copy of the file.

Left: Sets the left margin for the printed page. Units of measure (inches or centimeters) are those set in the Windows Control Panel. Margins do not change on the display screen.

Right: Sets the right margin for the printed page. Units of measure (inches or centimeters) are those set in the Windows Control Panel. Margins do not change on the display screen.

Top: Sets the top margin for the printed page. Units of measure (inches or centimeters) are those set in the Windows Control Panel. Margins do not change on the display screen.

Bottom: Sets the bottom margin for the printed page. Units of measure (inches or centimeters) are those set in the Windows Control Panel. Margins do not change on the display screen.

Font command button
Click this button to display the Printer Font dialog box, which lets you set a typeface and size for the printed output. The entire document, including headers and footers, uses the font you select. Settings become the default until you change them.

Printer Font dialog box

Use this dialog box to choose a typeface and size for your printed output.

Font list box: Select from the list of available fonts. Only fonts already loaded in your assigned printer appear. Use the Printer Setup command to choose a different printer. Use the Windows Control Panel to add additional fonts.

Size list box: Choose from the list of sizes available for the font you select.

Find Files Containing (Directory Browse) dialog box

This dialog box helps you search through multiple files. If you are not sure of the file or directory names, use the Directories and Drives list boxes to find the file you want.

Directory text box

Directories list box

Drives drop-down list box

Include subdirectories check box

Directory text box
Enter a drive and directory to specify where to search for files. If you are not sure of the directory name, select one from the Directories list box.

Include Subdirectories check box
Check this check box to include subdirectories in the search.

Compare dialog box

This dialog box lets you compare the contents of two files.

File 1 drop-down combination box

Line (File 1) text box

File 2 drop-down combination box

Line (File 2) text box

Horizontal option button

Vertical option button

Browse… command button

File 1 drop-down combination box
Enter the name of a file you want to compare. If you are not sure of the name or location of the file, use the Browse command button. Click the prompt button to choose from a list of files you have compared before.

Line (File 1) text box
Enter the line number in File 1 where you want to start the comparison. The default is to start at line 1.

File 2 drop-down combination box
Enter the name of a file you want to compare. If you are not sure of the name or location of the file, use the Browse command button. Click the prompt button to choose from a list of files you have compared before.

Line (File 2) text box
Enter the line number in File 2 where you want to start comparing. The default is to start at line 1.

Horizontal option button
Use this button to display the two files one above the other.

Vertical option button
Use this button to display the two files side by side.

Browse… command button
Click this button to display a standard browse dialog box, which lets you choose a file to compare. The filename you select appears in either the File1 or File 2 text box, depending on which text box is active.

Compare (Standard Browse) dialog box

Use this dialog box to find the files you want to compare. If you are not sure of the name, use the Files, Directories, Drives, and List Files of Type list boxes to find the file you want.

File Name text box

Files list box

Directories list box

List Files of Type drop-down list box

Drives drop-down list box

Find dialog box

Use this dialog box to find a text string in a single file.

NOTE: Find cannot locate a string that breaks over two lines. For the most efficient search, use the smallest unique portion of the text you want to find.

Pattern drop-down combination box

Match Upper/Lowercase check box

Regular Expression check box

Next command button

Previous command button

Pattern drop-down combination box
Enter the text you want to find. Click the prompt button to choose from a list of search strings you have used before. If you select (highlight) text in an active file, Find uses that text as the search string.

NOTE: To use regular expressions in the string, you must check the Regular Expression check box.

Match Upper/Lowercase check box
Use this check box to find an exact match for the search string.

When this check box is clear, the Desktop Editor considers any match to be a perfect match. For example, *GOTO* would match both *GoTo* and *goto*. To find only *GOTO*, enter the word in all caps in the Pattern drop-down combination box, and check the Match Upper/Lowercase check box.

Regular Expression check box
Check this check box to use regular expressions in the search string. Leave it unchecked if speed is important.

Next command button
When you have defined the search string, click this button to find the next occurrence of the string.

Previous command button
When you have defined the search string, click this button to search backward from the cursor.

Replace dialog box

Use this dialog box to specify text you want to find and text you want to replace it with. The search and replace always proceeds forward from the cursor position to the end of the document.

Search For drop-down combination box

Replace With drop-down combination box

Match Upper/Lowercase check box

Regular Expression check box

Confirm Changes check box

Search For drop-down combination box
Enter the text you want to find and replace. You can also click the prompt button for a list of search strings you have used before. If you check Regular Expressions, you can use regular expressions in the search string.

NOTE: Replace cannot find or replace a string that breaks from one line to another.

Replace With drop-down combination box
Enter the text you want to use as the replacement string. You can also click the prompt button for a list of replacement strings you have used before. You cannot use wildcards in this text box.

NOTE: If you leave this string blank, Replace deletes every occurrence of the search string.

Confirm Changes check box
Check this check box to be prompted before each change. When the check box is unchecked, the Desktop Editor automatically replaces the string each time it occurs. If this box is checked, the Confirm Replacement dialog box appears to confirm each replacement. The option buttons are Yes, No, and Cancel.

The Confirm Replacement dialog box also includes another Confirm check box. If you change your mind about confirming each replacement, and decide to simply replace every occurrence of the search string, uncheck the Confirm check box. When the prompt message changes to Replace All?, click Yes to continue with a global Replace.

Find Files Containing dialog box

Use this dialog box to specify:

- A text string you want to find
- Which drive and directories to search for files
- What type of files to look for

Pattern drop-down combination box

Directory static text field

Files drop-down combination box

Match Upper/Lowercase check box

Regular Expression check box

Directory command button

Directory static text field
Displays the current directory. To change the directory, click the Directory command button.

Files drop-down combination box
Enter the name of the file or files you want to search, or click the prompt button for a list of filenames or specifications you have used before. You can enter the names separated by spaces, or you can use one or more file specification wildcards.

You can include a drive ID and path in any file specification. If you do not, the default directory is assumed. To change the directory, use the Directory command button.

Directory command button
Click this button to select a default directory to search.

Goto Line dialog box

Use this dialog box to jump to a specific number in the file you are editing.

Line Number text box
Enter the line number you want to jump to. The range is from 0 to 32,365. The document scrolls to the line number you specify.

NOTE: If you highlight text in an active file and choose this command, the highlight extends to include the line number you specify.

List Found Files dialog box

After a Find Files Containing command finishes its search, the Desktop Editor displays this list of files found to match the search pattern. Highlight a filename and click Open to open the file.

Pattern field

Filter field

Files list box

Open command button

Pattern static text field
Displays the search pattern you specified with the Find Files Containing command.

Filter static text field
Displays the search criteria (other than the search pattern) you specified in the Find Files Containing command. This may be the specific filename or filenames, wildcard expressions, matching upper- or lowercase, or whatever you used to create this list.

Files list box
Lists the complete filename of every file matching the search criteria. Scroll through this list to select a file. If you open any of these files, the name disappears from the list.

Open command button
Opens the file you select in the Files list box.

Document Preferences dialog box

Use this dialog box to specify your preferences for certain document settings.

Tab Spacing text box

Right Margin text box

Word Wrap check box

Auto Indent check box

Expand Tabs with Spaces check box

Save as Default Settings check box

Tab Spacing text box
Sets the number of spaces between tab stops. The range is from 1 to 16.

Right Margin text box
Sets the right margin. The default is 65, and the range is from 32 to 512. This value is used only if you check the Word Wrap check box.

Word Wrap check box
Wraps text onto the next line at the right margin. A line ends without wrapping only when you press Enter.

Auto Indent check box
When you press Enter, the cursor starts the next line under the first nonblank character on the preceding line.

Expand Tabs with Spaces check box
Lets the Tab key indent the number of spaces you specified in Tab Spacing, but leave spaces instead of tab characters in the file.

Save as Default Settings check box
By default, you set options for this document only. Use this check box to apply these settings to all files in this and all future editing sessions.

Editor Preferences dialog box

Use this dialog box to specify your preferences for certain Desktop Editor functions.

Font group box

Cursor group box

Autosave Every X Minutes text box

Autosave Every X Changes text box

Undo Levels text box

Restore Session check box

Typing Replaces Selection check box

Make Backup Files check box

File Locking check box

Cut/Copy Current Line if No Text is Selected check box

Remove Trailing Spaces check box

Font group box
Click one of the option buttons to select the font type for this document. The actual screen font and size depend on settings in your SYSTEM.INI file.

System Fixed Font	Depends on FIXEDFON.FON
ANSI Fixed Font	Depends on FONTS.FON
OEM Fixed Font	Depends on OEMFONTS.FON

Cursor group box
Use the option buttons to select the shape of your cursor. Uncheck the Blinking check box if you do not want the cursor to blink.

Block	Highlights the current character.
Underline	Underlines the current character.
Vertical Bar	Uses the insertion point character as the cursor (default).
Blinking	Makes the cursor blink (default).

Autosave Every X Minutes text box
Enter the number of minutes you want. Enter 0 to turn the function off.

Autosave Every X Changes text box
Enter the number of edits you want. Enter 0 to turn the function off.

Undo Levels text box
Enter a number from 0 to 300. This is the number of edit changes you want the Desktop Editor to remember for the Undo function. Larger numbers increase the amount of memory that Desktop Editor needs. Setting this to 0 disables the function.

Restore Session check box
Check this check box if you want the Desktop Editor to reload the files that were loaded in the last session. By default, no files are loaded at startup.

Typing Replaces Selection check box
If you check this check box, highlighted text is replaced by the next character you type (such as a space) or the next Clipboard text you insert.
For example, this allows you to:

- Delete text by highlighting it, and pressing Spacebar.
- Replace a word by highlighting it and typing a new word.

Make Backup Files check box
Automatically makes a backup copy of your file every time you save it. A back-up file has the same name as the original, but with the extension .BAK.

File Locking check box
Check this check box if you want your files locked against use by others. For this to work, SHARE.EXE must be loaded before starting Windows, and a network must be active. If you use this option, you can have no more than 35 files open at one time.

Cut/Copy Current Line if No Text is Selected check box
If checked, the Desktop Editor selects the current line (wherever the cursor is at the moment) when you choose Cut or Copy.

Remove Trailing Spaces check box
If checked, the Desktop Editor deletes any extra spaces and tabs after the last character in every line.

Key Assignments dialog box

Use this dialog box to create keystroke shortcuts to speed up almost any Desktop Editor function.

Function combination box

Key combination box

Current Keys list box

Current Function list box

Enable Menu Accelerators check box

Keyboard Configuration File text box

Assign command button

Unassign command button

Load command button

Save command button

Browse command button

Function combination box
Enter a function name, in one of several ways:

- Type the function name directly into the text box.

Or,

- Scroll through the list using the scroll bar.

Or,

- Scroll through the list box using the DownArrow key.

As you move the focus, the current keystroke for that function, if any, appears in the Current Keys list box. Use the Assign and Unassign command buttons to assign keystrokes to functions.

Key combination box
Enter a keystroke name, in one of several ways:

- Type the function name directly into the text box.

Or,

- Scroll through the list using the scroll bar.

Or,

- Scroll through the list box using the DownArrow key.

As you move the focus, the current function for that keystroke, if any, appears in the Current Function list box. Use the Assign and Unassign command buttons to assign keystrokes to functions.

Current Keys list box
Displays the keys currently associated with the function highlighted in the Function text box. If there is no function assigned to a key or key combination, the box stays empty.

NOTE: You can assign the same function to more than one keystroke, to combine personal preference with general compatibility.

Current Function list box
Displays the function currently associated with the keystroke highlighted in the Current Keys list box. If there is no key assigned to that function, the box stays empty.

Enable Menu Accelerators check box
If this is checked (as it is by default) the standard menu accelerator keys take precedence over any assigned keystroke assignments.
If it is not checked, you can reassign a standard accelerator key (such as Alt+F, to drop down the File menu) to some other function.

Keyboard Configuration File text box
Displays the name of the keyboard file currently in use. \the efault is DEFAULT.KEY. If you want to store your custom keyboard in a separate file, enter a new name in this text box.

Assign command button
When you have selected a function and keystroke, click Assign.

Unassign command button
To disassociate a keystroke and a function, click Unassign.

Load command button
Loads a keyboard configuration file other than the one already listed in the Keyboard Configuration File text box.

Save command button
Click Save when you have finished assigning keystrokes and functions. Otherwise, changes will apply only for this session.

Key Assignments (Standard Browse) dialog box

This standard browse dialog box helps you locate or save keyboard configuration files. If you are not sure of the file or directory name, use the Files, Directories, Drives, and List Files of Type list boxes to find the file you want.

File Name text box

Files list box

Directories list box

List Files of Type drop-down list box

Drives drop-down list box

Configure Toolbar dialog box

Use this dialog box to choose how to display the toolbar.

Position group box: Lets you choose where you want the toolbar to appear on the ScriptMaker Editor window. The options are Top, Left, Right, or Bottom.

Style group box: Lets you choose how you want the toolbar to appear. The choices are Text only, Icon only, Text and Icon, or No Display.

Reference dialog box

Use this dialog box as a quick reference when you need to determine the exact syntax for a command, or when you would like to see an example of its use.

Commands list box

Description static text

Close command button

Add command button

Commands list box
Displays an alphabetical list of all ScriptMaker commands. Use ↑ and ↓ to scroll through the list, or press the first letter of any ScriptMaker command to move quickly to that part of the list.

Description static text
As you scroll through the Commands list, this area displays a short description, syntax, and example of each statement.

Close command button
Click this button to close the Reference dialog box.

Add command button
This inserts the currently highlighted statement from the Commands list into the text-editing screen at the current cursor position.
This handy feature helps avoid misspelling command names.

Insert Macro dialog box

Record group box

- Comments check box
- High-Level BASIC Statements check box
- Keyboard check box
- Mouse Relative To check box
- Screen option button
- Active Window option button

Comments check box
Adds a comment for each line of code generated during the recording. Each comment gives a brief explanation of what the corresponding line does.

High-Level BASIC Statements check box
Generates high-level BASIC statements corresponding to the occurrence of high-level events during the recording. The high-level statements control windows and dialog boxes and begin with **App** or **Win**. Maximizing a window and sizing a window are two examples of high-level events. The **AppMaximize** and **AppSize** statements replicate these events.

When this check box is not checked, only statements corresponding to the low-level events, such as mouse movements and mouse button presses, are generated. A single high-level statement for the event of sizing a window using the mouse translates to several low-level mouse events.

Keyboard check box
Generates statements corresponding to keyboard events. When this check box is not checked, no keyboard events, such as key presses/releases, are recorded.

Mouse Relative To check box
Generates statements corresponding to mouse events. Examples of mouse events are mouse movements and mouse button presses. Mouse events are recorded with the x and y position where they occurred. This position is recorded either as an absolute position of the screen in pixels or as a position relative to the active window in pixels, depending on whether you select the Screen or the Active Window option button, respectively. Selecting the Active Window option button generates a **QueSetRelativeWindow** statement (with a parameter of 0 to indicate the active window). Otherwise, mouse movements are relative to the screen.

Screen option button
Records mouse movements relative to the screen instead of to the position of the active window.

Active Window option button
Select to generate a **QueSetRelativeWindow** statement (with a parameter of 0 to indicate the active window). Otherwise, mouse movements are relative to the screen.

Stop Recorder dialog box

Place Recording group box

- Insert At Cursor Position option button
- Place Into Clipboard option button
- Insert As Main() option button

Insert At Cursor Position option button
Inserts the statements into your program at the current cursor position.

Place Into Clipboard option button
Copies the statements to the clipboard so you can copy the macro repeatedly.

Insert As Main( ) option button
Inserts the sub main statement, the recorded statements, and the end sub statements at the current cursor position.

Save Exe As

File Name text box
Files list box
Directories list box
List Files of Type drop-down list box
Drives drop-down list box

List Files of Type drop-down list box
Lets you quickly select a specific file type.

Extension	File Type
.EXE	Executable files
.	All file types in the directory

Save Code As dialog box

File Name text box
Files list box
Directories list box
List Files of Type drop-down list box
Drives drop-down list box

List Files of Type drop-down list box
Lets you quickly select a specific file type.

Extension	File Type
.SMC	Compiled Code files
.	All file types in the directory

Choose Icon dialog box

Icons box

Alternate Icon File drop-down combination box

Use Default command button

Browse... command button

Icons box
Displays the icon associated with the selected file or the default file, STUB.EXE. Whatever icon is currently selected is the icon for ScriptMaker .EXEs.

Alternate Icon File drop-down combination box
Displays the pathname to the file whose icon is displayed in the Icons box. You either type or browse for the name.

Use Default command button
Restores the default icon, a running man, associated with STUB.EXE file in the Norton Administrator console directory, to the Icons box.

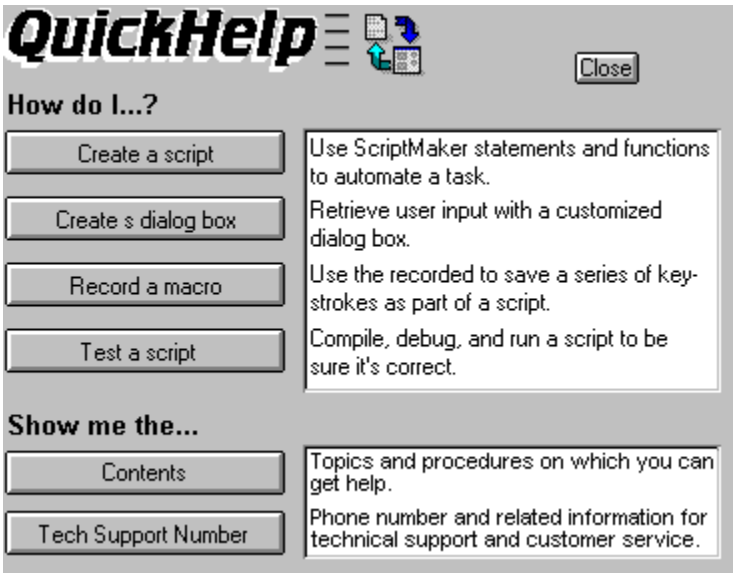
Browse... command button
Allows you to browse for an icon you want to use by selecting files that have icons. When the icon appears in the Icon box, you can decide whether or not you like it.

Choose Icon From... dialog box

File Name text box
Files list box
Directories list box
List Files of Type drop-down list box
Drives drop-down list box

List Files of Type drop-down list box
Lets you quickly select a specific file type.

Extension	File Type
ICONS	Files that have icons
.	All file types in the directory


QuickHelp
Close
How do I...?
Create a script
Use ScriptMaker statements and functions to automate a task.
Create s dialog box
Retrieve user input with a customized dialog box.
Record a macro
Use the recorded to save a series of key-strokes as part of a script.
Test a script
Compile, debug, and run a script to be sure it's correct.
Show me the...
Contents
Topics and procedures on which you can get help.
Tech Support Number
Phone number and related information for technical support and customer service.

ScriptMaker Help Contents

Getting Started

Procedures

ScriptMaker Help Contents

Getting Started

Procedures

ScriptMaker Help Contents

Getting Started

Procedures


Testing a Script


Halting Script Execution in the Editor


Saving a Compiled Script


Saving a Script as an Executable File


Executing a Script from the Command Line


The ScriptMaker Editor

ScriptMaker Help Contents

Getting Started

Procedures

ScriptMaker Help Contents

Getting Started

Procedures

ScriptMaker Help Contents

Getting Started

Procedures

ScriptMaker Help Contents

Getting Started

Procedures

ScriptMaker Help Contents

Getting Started

Procedures

ScriptMaker Help Contents

Getting Started

Procedures

Working with Files

Opening a File

Inserting One File into Another

Moving and Copying Text

Moving to a Specific Line in a File

Comparing Two Files

Printing All or Part of a File

Using Multiple Edit Windows

Using ScriptMaker Editor Macros

Saving Files

Saving a File with a New Name

Saving Text as a Separate File

Mouse and Keyboard Operations

See Also

The ScriptMaker Editor lets you assign your own favorite keystrokes to any of its over 100 editing functions. See Customizing Editor Keystrokes for details.

You can customize the entire ScriptMaker Editor keyboard. For a complete list of Editor key functions, refer to Editor Functions.

Cursor Motion Key Defaults

See Also

Key	Default Editor Function
←	cursor_left
Ctrl+ ←	word_left
→	cursor_right
Ctrl+ →	word_right
↑	cursor_up
Ctrl+ ↑	window_down
↓	cursor_down
Ctrl+ ↓	window_up
PgUp	page_up
Ctrl+PgUp	top_of_window
PgDn	page_down
Ctrl+PgDn	bottom_of_window
Home	beginning_of_line
Ctrl+Home	beginning_of_buffer
End	end_of_line
Ctrl+End	end_of_buffer
Tab	tab_right
Enter	enter
Ctrl+G	goto_line

Text Selection Key Defaults
F1…F12 Function Key Defaults
Other Key Defaults

Text Selection Key Defaults

See Also

Key	Default Editor Function
Shift+ 	select_line_up
Shift+ 	select_line_down
Shift+ 	select_char_left
Shift+Ctrl+ 	select_word_left
Shift+ 	select_char_right
Shift+Ctrl+ 	select_word_right
Shift+PgUp	select_page_up
Shift+PgDn	select_page_down
Shift+Home	select_to_bol
Shift+Ctrl+Home	select_to_top
Shift+End	select_to_eol
Shift+Ctrl+End	select_to_end
Esc	unmark_block

NOTE: The default keys are assigned so that pressing Shift along with a cursor motion key selects text as the cursor moves.

F1...F12 Function Key Defaults

See Also

Key	Default Editor Function
F1	editor_help
F2	save_file
F3	open_file
F4	document_preferences
Ctrl+F4	close_window
F5	(none)
Ctrl+F5	restore_window
F6	(none)
Ctrl+F6	next_window
F7	record_macro
F8	play_macro
F9	(none)
F10	(none)
Ctrl+F10	zoom_window
F12	wrap_para

Cursor Motion Key Defaults
Text Selection Key Defaults
Other Key Defaults

Other Key Defaults

See Also

Key	Default Editor Function
BkSp	backspace
Alt+BkSp	undo
Ctrl+BkSp	delete_word_left
Del	delete
Ctrl+Del	delete_word_right
Shift+Del	cut
Ins	toggle_insert
Ctrl+Ins	copy
Shift+Ins	paste
Enter	enter
numpad *	undo
numpad +	copy
numpad -	cut
Tab	tab_right
Alt+ -	close_window
Alt+D	delete_line
Alt+K	delete_to_eol
Alt+N	next_window
Alt+P	print
Alt+V	about
Alt+X	save_all_exit

Cursor Motion Key Defaults
Text Selection Key Defaults
F1…F12 Function Key Defaults

Launching ScriptMaker

See Also

The ScriptMaker Editor is a text editor used for creating ScriptMaker scripts. The ScriptMaker language is similar to BASIC, designed for use in the Windows environment.

To start the ScriptMaker Editor:

- Choose ScriptMaker from the Norton Desktop Tools menu.

What is a Script?

See Also

A script file is simply a list of instructions for the computer to process. Any task that you plan to run more than once, or that requires entering the same keystrokes, is a candidate for a script file. The statements you enter in your script file are executed in sequence when you run the script.

Once you have written your script using the ScriptMaker Editor, you compile and test it to be sure it is accurate. Then you save the compiled code or create an executable file. Compilation is the process of converting the statements entered in your script (.SM) into a low-level set of commands that can be executed, or run. ScriptMaker also checks for syntax errors while your script compiles. When you save the compiled code, the extension to use is .SMC. If you save the compiled code as an executable file, some additional execution information is added to the code, and the extension to use is .EXE.

Creating a Script

See Also

Use the commands on ScriptMaker's File menu to create new script files and modify existing scripts. To create a new script that is similar to an existing one, you can use the existing script as the basis for the new one.

To create a new script file:

1. Choose New from the File menu.
   A new, untitled script window appears. (If you have just entered the Editor, you already have this window.)
2. Simply start typing ScriptMaker statements in the editing window.
3. When you are finished, choose Save As from the File menu to store the script file in a text file.
   The Save As dialog box that appears.
4. Enter the name of your script file in the File Name text box. Use the extension .SM.
   To store the script file in a directory other than the current one, you can use the directory and drive list boxes to specify the desired directory.

TIP: If you are creating a fairly long script file, use the Save As command to assign the filename while you are still working on the file. Then choose Save from the File menu to save the file from time to time, as well as when you are finished creating it. These methods keep you from losing your work in the event of a power failure or any other problem that causes ScriptMaker to close prematurely.

Editing a Script

See Also

Once you have created, compiled, and run a script, you may find that there are changes you want to make. ScriptMaker makes it easy to modify an existing script.

To edit an existing script file:

1. Choose Open... from the File menu.
   The Open File dialog box appears.
2. Select the script file you want to edit from the File Name list box.
3. Click OK.
   The script you have selected appears in a window in the Editor.
4. Edit your script file.
5. Choose Save from the File menu to save your changes.

TIP: To prevent possible loss of any of your work, you may want to use the Save command every few minutes while youre editing the script.

Creating a New Script from an Existing One

See Also

In some cases, a script you have written may be similar to the one you want to create. Use ScriptMaker to enhance an existing script and save it under a new name.

To create a new script from an existing one:

1. Choose Open... from the File menu.
   The Open File dialog box appears.
2. Select the existing script you want to use as a basis for your new script from the File Name list box.
3. Click OK.
   The selected script appears in a window in the Editor.
4. Choose Save As... from the File menu.
   The Save As dialog box appears.
5. Type the filename you want assigned to your new script file in the Save As dialog box. Use the extension .SM.
   Because you now have a new copy of the original script file, you can make changes to this new file without affecting the original.
6. Enter the desired changes to the lines in this new script file.
7. Choose Save from the File menu to save the file from time to time during your editing session, and to save it after youve entered all your edits.

Inserting a Macro into a Script

See Also Dialog Box Settings

When you click the End button on the Recorder, you stop recording and the Stop Recorder dialog box appears.

1. Select the option button that indicates where and how you want to insert or copy the ScriptMaker statements created by the Recorder.
   - Insert at cursor position. (This inserts the optimized code for the macro at the cursor position.)
   - Place into clipboard. (This inserts the optimized code for the macro into the clipboard.)
   - Insert as Main( ). (This inserts the optimized code for the macro at the cursor position enclosed in the statements **Sub Main** and **End Sub**.)
2. Click OK to accept the macro.

Using the Macro Recorder

Inserting a User-Defined Dialog Box Into a Script

See Also

To use a dialog box template in a script that displays one or more instances of that dialog box, you must:

- Declare the dialog box template: Adding the text version of a dialog box template (the lines that start with **Begin Dialog** and end with **End Dialog**) to the script creates a template declaration for that dialog box in the script. You can declare more than one template per script (so long as they have different names).
- Display the dialog box for the user: Declare a variable as an instance of the dialog box template and use the **Dialog** function to display that instance.
- Make adjustments to the script to accommodate the dialog box template:
  - Before the Begin Dialog statement: Declare the variables that appear in the template and provide values for those variables.
  - Between the End Dialog statement and the Dialog function: Preset the values of any of the components (called controls).
  - After the Dialog function: Find out what command button the user selected to close the instance of the dialog box and determine the values of the other components of the dialog box.

The Component Interaction with Script chart shows what the script can send to and receive from each type of dialog box component.

Displaying a User-Defined Dialog Box

See Also Example

To display the dialog box is simply a matter of declaring an instance of the dialog box and using the **Dialog** function. This declaration must come after the template declaration (**Begin Dialog...End Dialog**).

Syntax:

Dim *instanceName* **As** *templateName*
selectedButton = **Dialog** (*instanceName*)

instanceName	An identifier used as the name of this instance of the dialog box template.
templateName	The identifier used as the name of the template in its declaration (the word after **Begin Dialog**).
selectedButton	Numeric variable for the number of the command button selected by the user (which is what the **Dialog** function returns).

NOTE: If the only button is the OK button, you can use the dialog statement whose syntax is **Dialog** instanceName.

Displaying Dialog Box Example

This example displays an instance of the template named UserDialog. The name of the instance is MyDialog.

```
'declares variable for command button
'number returned by dialog function
Dim Button_Number%
...
'template declaration
Begin Dialog UserDialog ...
   ...
End Dialog
...
'declaration of instance of template
Dim MyDialog As UserDialog
...
'displays dialog box
Button_Number = dDialog (MyDialog)
...
```

Declaring Template Variables and Their Values

See Also Example

If the template has variables in it, the program does not compile until you declare those variables. These variables are all strings or string arrays. You must declare them BEFORE the **Begin Dialog** statement. You must also give them values or the variables will be the empty string ("") and the arrays will have an empty string in each element.

Look for the following possible variables:

- The label for the dialog box (which appears in its title bar, but is defined at the end of the **Begin Dialog** statement).
- The labels for command buttons, check boxes, group boxes, and option buttons. The label always follows the height in the statement that defines the component.
- Static text. The text appears at the end of the Text statement in the template. When next to a text box, list box, or combination box, static text is considered the label for that box.

Each of these labels can be either a string literal (enclosed in double quotation marks) or a variable. It the label is a string literal, it was predefined in the Dialog Editor and cannot be changed by the script. If the label is not in quotation marks, you declare it exactly as it appears in the template. The following are examples of check box declarations.

```
CheckBox 126,94,108,14, "Use Low Calorie Ingredients", .LowCalorie
CheckBox 126,111,48,14, Charge, .PaymentType
```

The labels are "Use Low Calorie Ingredients" and Charge. Charge is the only variable name. It has to be declared before the **Begin Dialog** statement.

```
Dim Charge as string
Charge = "Cash"
```

Or,

```
Charge$ = "Cash"
```

The following always have variables that you must declare:

- List boxes.
- Combination boxes.

Each list box and combination box declaration always contains a variable for a string array. The array name always follows the height in the box's declaration. The next example shows the declaration of a combination box.

```
ComboBox 16,38,42,28, EntreeArray, .Entree
```

The string array, EntreeArray, has to be declared and should be filled before the **Begin Dialog** statement.

```
Dim EntreeArray$ (1 to 3)
```

NOTE: As you fill the arrays for list boxes and combination boxes, put the most likely choices first to save the user time. The first choice is always selected by default. (The first option button is the default, too.) Don't skip subscripts, because an empty line (which can be selected) will appear in the boxes for each missing subscript.

Declaring and Setting Template Variables Example

This example shows the declaration and assignment statements for the check box and combo box mentioned in this section.

```
'declares variable for command button
'number returned by Dialog function
Dim Button_Number%
'declares variable for check box label
Dim Charge$
'label for check box
Charge = "Use Charge Card"
'declares variable for array to fill combo box
Dim EntreeArray$ (1 To 10)
'values for array
EntreeArray (1) = "Sword Fish"
EntreeArray (2) = "Crab Legs"
EntreeArray (3) = "Eggplant Parmesan"
...
'template declaration
Begin Dialog UserDialog ...
   ...
End Dialog
'declaration of instance of template
Dim MyDialog As UserDialog
...
'displays dialog box
Button_Number = Dialog (MyDialog)
```

Presetting the Values of Dialog Box Components

See Also Example

Between the declaration of the instance and the **Dialog** function, you can preset:

- The value of a check box to checked or unchecked.
- An option button selection from a group of option buttons.
- The item selected in a list box or combination box.

For example, to save the user time, you may want to preset a component to the value it is most likely to have. If a check box is very likely to be checked by the user, you can set it to be checked from the script. The user can uncheck it on the few occasions when the check box isn't wanted.

When you preset the value of a component, you use the name of the dialog box instance and the field name for the component. The period, which always appears at the beginning of the field name, acts as a separator between the two names. The field name (when a component has one) is always the last item in the declaration. For example, in the following CheckBox declaration, .PaymentType is the field name.

```
CheckBox 126,111,48,14, Charge, .PaymentType
```

In the following example, the instance is MyDialog, the check box's field name is .PaymentType, and the check box is preset to be checked for the user.

```
'template declaration
Begin Dialog UserDialog ...
   ...
End Dialog
'declaration of instance of template
Dim MyDialog as UserDialog
...
'presets check box to checked
MyDialog.PaymentType = 1
...
'displays dialog box
Button_Number = dialog (MyDialog)
```

NOTE: You cannot preset components before the template declaration, because the field names are declared only by their appearance in the template.

Presetting the Values of Dialog Box Components Example

In the following definition of a group of option buttons, the field name is .IceCream.

```
OptionGroup .IceCream
   OptionButton 30, 112, 48, 14, "Vanilla"
   OptionButton 30, 126, 48, 14, "Chocolate"
   OptionButton 30, 142, 48, 14, "Strawberry"
```

By default, the first option button (in this case "Vanilla") is always preset. The following example presets "Chocolate" instead. .IceCream is a numeric field that numbers the option buttons in the order they appear in the template, starting with zero. The option button whose label is "Chocolate" is number 1.

```
'template declaration
Begin Dialog UserDialog ...
   ...
End Dialog
'declaration of instance of template
Dim MyDialog as UserDialog
...
'presets option button to "Chocolate"
MyDialog.IceCream = 1
...
'displays dialog box
Button_Number = dialog (MyDialog)
```

Obtaining Output from Dialog Box Instance

See Also Example

After the program displays the dialog box using the **Dialog** function, it waits for the user to click one of the command buttons. Obviously, you displayed the dialog box because you wanted the data from it, so you must retrieve it.

It is very important to find out what command button the user selected. If it was Cancel, you don't want to retrieve any data. The values of the fields remain as they were prior to the **Dialog** function. If the number of the button returned by the function is 0, the user selected Cancel.

For all the data retrieved from the dialog box other than the command button's number, you use the instance name for the dialog box in combination with the field name for the component. The instance name appears in the **Dim** statement that declares the instance and in the **Dialog** function call. The field name is the name that starts with a period in the declaration of the component (between the **Begin Dialog** and **End Dialog** statements).

You will want to process the data in some way after you retrieve it. The processing is not discussed here because it varies greatly from program to program.

Obtaining Dialog Box Output Example

This example shows the retrieval of the string returned from a combination box and the number (1 for checked or 0 for unchecked) returned from a check box.

```
EntreeChoice$
TypeOfPayment%
...
'displays dialog box
Button_Number = Dialog (MyDialog)
If Button_Number <> 0 Then
   'retrieve data from dialog box
   EntreeChoice = MyDialog.Entree
   TypeOfPayment = MyDialog.PaymentType
   ...
End If
```

How Components Interact with Script

See Also

Component	Sent from Script	Returned to Script
Name that appears in the dialog box title bar	The name can be a value of a string variable from the script or predefined in the template.	N/A
Command button	The label on the button can be the value of a string variable from the script or predefined in the template.	Number. Command buttons (other than OK and Cancel) are numbered (starting from 1) in the order they are created (or, ultimately, appear in the template declaration). The button number is returned by the **Dialog** function.
OK Button	N/A	Number. Returns the number -l from the **Dialog** function.
Cancel Button	N/A	Number. Returns the number 0 from the **Dialog** function.
Text Box	N/A	String. Returns the characters in the text box.
Text	The text can be the value of a a string variable from the script or already defined in the template.	N/A
Group Box	The label can be the value of a string variable from the script or already defined in the template.	N/A
Option Button	The label can be the value of a string variable from the script or already defined in the template.	Number. Option buttons are numbered per set (starting from 0) and in the order they were created (or, ultimately, appear in the template declaration). The button number is returned.
Check Box	The label can be the value of a string variable from the script or already defined in the template.	Number. When checked, a 1 is returned. When not checked, 0 is returned.
List Box	The values in the list box come from a string array. Empty elements appear as empty items in the list, so it is best not to skip subscripts. The name of the array is the name you put in the Array$ text box in the template when defining the list box.	Number. Returns the subscript for the array item that was selected.
Combo Box	Values in combo box come from a string array. Empty elements are displayed as empty items in the list, so it is best not to skip subscripts. The name of	String. Returns the string value of a the selection. If the user types in a string, it is returned instead of one of the strings in the array that filled the list.

the array is the name you
put in the Array$ text box
in the template when
defining the combo box.

If the user cancels the dialog box, the variables that would have received values from the dialog box components keep the values they had before the dialog box appeared.

Compiling a Script

See Also

To check the syntax of the statements in your script, run the Compiler. The compiler returns the error number and message for the first error it finds along with the line number of the offending statement. You correct the error and recompile. When the script is syntactically correct, the message Compile Successful appears and you can execute the script.

To find the first syntax error in a script:

- Choose Compile from the Script menu.

Executing a Script from the Editor

See Also

To test whether your compiled script does what you intended, run it in the Editor. If you have a problem, you can halt the script's execution.

To run your script from the Editor:

- Choose Run from the Script menu.

Halting a Script in the Editor

See Also

Sometimes you do not want to complete the script's execution, in which case you can abort or halt it.

To halt execution:

- Choose Abort from the Script menu when the script pauses for input; for example, when a message box is displayed.

Saving a Compiled Script

See Also

You can save a compiled script and run it from the command line as a parameter for SMW.EXE, the version of ScriptMaker that executes in Window or convert the script into an executable file with the .EXE extension.

To save a compiled script:

1. Choose Compile from the Script menu and compile the script successfully.
2. Choose Save Code from the Script menu.
   The Save Code As dialog box displays the name of the script with an .SMC extension in the File Name text box.
3. Change the drive or directory, if necessary, and click OK.

Saving a Script as an Executable File

See Also Dialog Box Settings

You can convert scripts into executable files with an .EXE extension.

To save a script as an executable file:

1. Choose Compile from the Script menu and compile the script successfully.
2. Choose Save Exe from the Script menu.
   The Save Exe As dialog box displays the name of the script with an .EXE extension in the File Name text box.
3. Change the drive or directory if necessary.
4. If you want an icon to appear when the file is on the desktop or is minimized, click Icon.
   The Choose Icon dialog box appears.
5. In the Alternate Icon File drop-down combination box, type the name of a file that has an icon you would like to use or click Browse... and use the Choose Icon From... dialog box to locate a file. As you make your selection, the icon (or icons) appear in the Icon box.
   Click Default to restore the default icon to the Icon box.
6. Click OK to return to the Save Exe As dialog box.
7. Click OK to return to the ScriptMaker Editor.

NOTE: Most applications that execute in Windows have an icon you can "borrow." You cannot use a bitmap file as you would for Windows wallpaper.

Executing a Script from the Command Line

See Also

You can execute scripts as compiled code (.SMC) or an executable file (.EXE) from the command line.

To execute compiled code:

- Use ScriptMaker's execution program (SMW.EXE) along with the complete or relative pathname to the script. You must use the .SMC extension. For example, the following launches the compiled file ADDAPP.SMC in the current directory:
  SMW ADDAPP.SMC

To execute an executable file:

- Use the name of the executable file on a command line.

To halt execution:

- Click the icon for the script, and choose Close from the Control menu.

Customizing ScriptMaker Editor

See Also Dialog Box Settings

You can customize ScriptMaker Editor to suit the way you work. If you are already familiar with other editors or word processors, you'll find it easy to do the things you want to do. As you become more comfortable with ScriptMaker Editor, you may want to make further adjustments to speed up common operations.

Everything you need to customize ScriptMaker Editor is available in the Options menu. You can set:

- Editor preferences, such as automatic backup and file saving options.
- Key combination assignments.
- Toolbar display.
- Document preferences such as tab spacing and margins.

When you chose Customize... from the Options menu, the left side of the screen displays an icon for each of the four Options categories. Once you have finished with one set of changes, just click the next icon to move on to the next.

Setting Document Preferences

See Also Dialog Box Settings

The features you set here, such as tab spacing, margin size, and word wrap, apply to all your ScriptMaker Editor files.

To set or change document preferences:

1. Choose Document Preferences… from the Options menu,
   The Document Preferences dialog box appears.
2. Set each option as desired.
3. Check the Save As Default check box to use these settings for all your documents.
4. Click OK.

TIP: To toggle word wrap on and off, choose Word Wrap from the Edit menu. To reformat a paragraph so that it wraps correctly after editing, choose Wrap Paragraph from the Edit menu.

Setting Editor Preferences

Setting Editor Preferences

See Also Dialog Box Settings

These settings control how the ScriptMaker Editor manages the files you edit. The features you set here apply to all your files.

To change Editor preferences:

1. Choose Customize… from the Options menu,
   The Editor Preferences dialog box appears.
2. Set each option, as desired.
3. Click OK or select another category from the Categories list box. Either way, your changes are saved.

Setting Document Preferences

Customizing the Keyboard

See Also Dialog Box Settings

You can customize your keyboard to put complex operations on a single key combination, or simply to make the ScriptMaker Editor keystrokes more like a text editor you already know. There are more than 100 ScriptMaker Editor functions available.

NOTE: For a complete list of ScriptMaker Editor key functions, refer to Editor Functions.

To assign a function to a keystroke:

1. Choose Customize… from the Options menu,
   The Editor Preferences appears.
2. Select Key Assignments from the Categories list box.
   The Key Assignments dialog box appears.
3. Scroll through the Function list box until you highlight the function you want to assign.
   The name now appears in the Function text box. If it is already assigned to a keystroke, and the key names appear in the Current Keys list box. If no key is assigned, the box is empty.

   NOTE: You can assign a function to more than one keystroke. For example, both **Ctrl+C** and **Ctrl+Insert** can copy text from the Clipboard.
4. Scroll through the Key list box until you find the keystroke you want to use.
   If the key is already being used for some other function, the function name appears in the Current Function list box. If no function is assigned to that keystroke, the box stays empty.

   NOTE: You cannot assign a keystroke to more than one function. For example, the only function **Ctrl+C** can perform is to copy text from the Clipboard.
5. Click Assign to make the assignment.
6. Click Save to make the change permanent. Otherwise, your key assignments last only for this session.
7. Click OK or select another category from the Categories list box.

To change a keystroke assignment:

1. Choose Customize from the Options menu.
   The Editor Preferences dialog box appears.
2. Select Key Assignments from the Categories list box.
   The Key Assignments dialog box appears.
3. Scroll through the Key list box until you find the keystroke you want to reassign.
   The function name currently assigned to the keystroke appears in the Current Function list box.
4. Use the Function list box to highlight a different ScriptMaker Editor function.
5. Click Assign.
6. Click Save to make the change permanent.
7. Click OK or select another category from the Categories list box.

To delete a keystroke assignment:

1. Choose Customize from the Options menu.
   The Editor Preferences dialog box appears.
2. Select Key Assignments from the Categories list box.
   The Key Assignments dialog box appears.
3. Find the key/function combination in the Key and Function list boxes.
4. Click Unassign.
5. Click Save to make the change permanent.
6. Click OK or select another category from the Categories list box.

Looking Up Key Assignments

Customizing the Toolbar

See Also Dialog Box Settings

You have several options for using the toolbar, including whether or not you want to use it at all. The features you set here apply to all your files.

To toggle the Toolbar on or off:

- Choose Toolbar from the Options menu.

You can also use the Customize... command to determine whether or not the Toolbar is displayed.

To customize the Toolbar:

1. Choose Customize… from the Options menu.
   The Editor Preferences dialog box appears.
2. Select Toolbar from the Categories list box.
   The Toolbar dialog box appears.
3. Set each option, as desired. To disable the toolbar, select No Toolbar from the Style group box.
4. Click OK or select another category from the Categories list box. Either way, your changes are saved.

Setting Up Your Printer

Dialog Box Settings

Use ScriptMaker's printer options to specify settings such as margins, headers, and footers, to get the printing results you want.

To set up your print specifications:

1. Choose Printer Setup from the File menu to specify a printer.
2. Choose Page Setup from the File menu to use the Page Setup dialog box.
3. Select printing options, including margins, headers, and footers.
4. Click Font to select a printer font from the Printer Font dialog box.

   NOTE: Only fonts already loaded in your assigned printer are available.

Page Setup dialog box
Printer Font dialog box

Setting Up the Printed Page

Dialog Box Settings

You adjust the page layout for your document with the Page Setup dialog box.

To set up the page layout:

1. Type a line of text to be centered at the top of each page in the Header combination box or select a past header.
2. Type a line of text to be centered at the bottom of each page in the Footer combination box or select a past footer.

   NOTE: You can use %p for a page number, %d for the time and date, and %f for the complete pathname to the document file.
3. Type the margins you want to use in the Left, Right, Top, and Bottom text boxes.
   The margins are in inches or centimeters depending on the setting in your Windows Control Panel.
4. Click Font to select a printer font from the Printer Font dialog box.

   NOTE: Only fonts already loaded in your assigned printer are available.
5. Click OK to return to the Editor.

Page Setup dialog box
Printer Font dialog box

Selecting a Font

Dialog Box Settings

The Editor uses the font you select as the default font for all print jobs.

To select a font:

1. Select a font from the Font list box.
2. Select a size from the Size list box.
3. Click OK.

Searching for Text in a File

See Also Dialog Box Settings

The Find and Find Again commands search for character strings in a single file. The search begins at the cursor and ends at the end of the file, if you are searching forward using the Next button. You can also search from the cursor backward to the beginning of the file using the Previous button.

NOTE: Find cannot locate a string that breaks over two lines. For the most efficient search, use the smallest unique portion of the text you want to find.

To search for text:

1. Choose Find from the Search menu
   Or,
   Click the Find button
   The Find dialog box appears.
2. Enter the text you want to find in the Pattern text box.
   Or,
   Use the Pattern drop-down list box to choose from search strings you have used recently.

   NOTE: If you select text in the file and then choose Find, the selected text appears automatically in the Pattern text box.
3. Check the Match Upper/Lowercase check box if you want the search to be case-sensitive.
4. Check Regular Expression if you are using regular expressions in the search string.

   NOTE: If you are not using regular expressions, make sure this check box is blank to speed up the search.
5. Click Next to start the search forward, or click Previous to search backward.
   If ScriptMaker Editor finds a match, it displays and highlights the text. Otherwise, "Pattern Not Found" appears in the status line.

To continue a search:

- Choose Find Again from the Search menu.

Searching for Text in Multiple Files

See Also Dialog Box Settings

ScriptMaker Editor lets you search all the files in a single directory (but not all the directories on the disk) for a particular text string. For example, you may want to modify all the instances of a variable in your programs. Or you may decide to change the name of the main character in your multi-part novel.

NOTE: Find cannot locate a string that breaks over two lines. For the most efficient search, use the smallest unique portion of the text you want to find.

To find files containing a particular text string:

1. Choose Find Files Containing... from the Search menu to display the Find Files Containing dialog box.
2. Enter the search string in the Pattern text box, or use the drop-down list box to choose from a list of strings you have used before.

   NOTE: You can use regular expressions in your search string. For details, see Searching with Regular Expressions

   ---
3. Specify the directory you want to search. Click Directory if the directory name displayed is not the one you want.
4. Enter the filenames you want to search in the Files text box:
   - Enter one or more filenames separated by spaces; for example, `summer.txt. test1.bat`
     Or,
   - Enter one or more wildcard specifications, separated by spaces;
     for example, `*.txt *.bat.`
     Or,
   - Use the drop-down list box for file specifications you have used before.
5. Check Match Upper/Lowercase if you want the search to be case-sensitive.
6. Check Regular Expression if you are using regular expressions in the search string.
7. Click OK to start the search.
8. If ScriptMaker Editor finds the string, a list of filenames appears in the List Found Files dialog box.
9. To open a file, double-click any filename or select the file and click Open.
   The first occurrence of the search string is highlighted when the file opens.

Selecting a Search Directory

When you are searching multiple files, you can specify the path for the search using the Select Directory dialog box.

To select a search directory:

1. To change drives, select a drive from the Drives drop-down list box.
2. To change directories, select a directory from the Directory list box.
3. Click OK to return to the Find Files Containing dialog box.

Opening a Found File

You can check the contents of the files you have found.

To open a file:

1. Double-click the name of a file in the Files list box.
2. If this is not the file you want after all, choose List Found Files from the Options menu and repeat step 1.

Finding and Replacing Text

See Also Dialog Box Settings

The search and replace feature lets you find a specified text string in a file and replace it with another string. By default, the ScriptMaker Editor prompts you to confirm each substitution, so you can skip a replace when the change would not be appropriate.

The replace operation always starts from the cursor and moves forward through the file. To be thorough, make sure the cursor is at the top of the file before beginning the replace.

To replace a character string:

1. Choose Replace from the Search menu to display the Replace dialog box.
2. Enter a character string in the Search For text box.
   Or,
   Choose a string you have used before from the Search For drop-down list box.
3. Enter a character string in the Replace With text box.
   Or,
   Choose a string you have used before from the drop-down list box.
4. If you want the Replace to be case-sensitive, check the Match Upper/Lowercase check box.
5. If you are using regular expressions in the string, check the Regular Expressions check box.
6. Check the Confirm Changes check box if you want to be prompted before each replace.
7. Click OK.
   A Confirm Replacement dialog box appears before every change. Click Yes to confirm or No to go on to the next change. Click Cancel to stop the replace.

   NOTE: If you have started the replace operation with Confirm Changes and decide to let the ScriptMaker Editor continue changing globally, uncheck the Confirm check box in the Confirm Replacement dialog box.

Searching for Text in a File

Searching with Regular Expressions

See Also Regular Expressions Examples

The Find, Replace, and Find Files Containing commands all allow you to search for text using text wildcards called regular expressions. This is an extremely flexible wildcard feature that, among other things, allows you to find:

- Text in formatted layouts, such as phone numbers, quoted strings, text in parentheses, and so on
- Programming terms and expressions
- Proper names or the start of a sentence (that is, any string with the first letter capitalized)
- Blank lines
- A sequence of digits
- Lines that begin or end with the specified text
- Alternate spellings of a word or name
- Leading or trailing blanks in a line

To define a search using regular expressions:

1. Choose Find, Replace, or Find Files Containing from the Search menu to display the appropriate dialog box.
2. Use one or more of the expressions in the Regular Expressions table when you enter the search string in the Pattern text box.
   Click *Regular Expressions* at the top of this Help window to display the table.
3. Check the Regular Expression check box.
4. Click Next (in Find) or OK (in Replace or Find Files Containing) to start the search

You can also use this feature to select not merely the search string, but the whole line in which it appears. For example, <*goto*> finds the string "goto" and the entire line that contains it.

Click *Regular Expressions* at the top of this Help window to see characters allowed in searches. Click *Examples* to see some sample uses of these characters.

Regular Expressions

Regular Expression	Matches
?	Any single character.
*	Zero or more occurrences of any character.
@	Zero or more occurrences of the previous character or expression.
% or <	The beginning of a line.
$ or >	The end of a line.
%$ or <>	A blank line.
\t	A tab character.
\f	A form feed character.
\	The next character literally, rather than as a wildcard. Not required if you are not using regular expressions. For example, * finds an asterisk, and \\ finds a backslash.
[]	Any of the characters between the brackets. Use a hyphen to specify a range of characters. For example, [abc] matches a, b, or c, and [A-Za-z] matches any upper- or lowercase letter.
[~]	Any character *except* those between the tilde and the right bracket. Use a hyphen to specify a range of characters. For example, [~abc] matches any character except a, b, or c, and [~A-Za-z] matches any non-alphanumeric character.

Examples

This pattern…	Finds…
(KC)athy	Kathy or Cathy
(KC)athy (A-Z) ~ @	Kathy or Cathy and her last name
(209)??-(0-9)???	Any phone number
<*(209)??-(0-9)???*>	Any line that contains a phone number
<*(A-Za-z)@(*){>	A line that defines a C function, such as int Dolt(x,y){
<{	Finds { when at the start of a line
<(\t)(\t)>	Any line containing only blanks or tabs
\\(nrt)	Any occurrence of \n, \r, or \t

Looking Up Key Assignments

See Also Dialog Box Settings

There are more than 100 editing functions available in ScriptMaker Editor. Some of their key assignments may have been reassigned, or you may have reassigned them yourself. You may find it useful to look through the Key Assignments list just to find some helpful shortcuts.

To see what key performs what function:

1. Choose Customize from the Options menu.
   The Editor Preferences dialog box appears.
2. Select Key Assignments from the Categories list box.
   The Key Assignments dialog box appears.
3. Scroll through the Function list box until you find the function you are interested in.
   The name appears in the Function text box, and any associated keystrokes appear in the Current Keys list box below it.
4. Click Cancel to avoid changing any settings.

NOTE: For a complete list of ScriptMaker Editor key functions, refer to Editor Functions.

Customizing the Keyboard

Opening a File

See Also Dialog Box Settings

There are several ways to open a file in ScriptMaker Editor. In some cases, you do not have to start ScriptMaker Editor first.

To open a file in ScriptMaker Editor:

1. Choose the Open command from the ScriptMaker Editor File menu.
   Or,
   Click the Open button.
   The Open File dialog box appears.
2. Enter the name of the file you want to open in the File Name text box.
   If you are not sure of the filename, use the Drives, Directories, and List Files of Type list boxes to find it.
3. Click OK.

Or,

- Drag a file from a drive window into the ScriptMaker Editor application window or into its minimized application icon.
  The icon automatically maximizes and opens ScriptMaker Editor.

Or,

1. Open the ScriptMaker Editor File menu
2. Choose from the four recently edited files listed at the bottom of the menu.

To automatically load and open the same set of files the next time you start ScriptMaker Editor:

1. Choose Customize… from the Options menu,
   The Editor Preferences dialog box appears.
2. Check the Restore Sessions check box.
3. Choose Exit from the File menu, without closing the files individually.
   The next time you start ScriptMaker Editor, it automatically loads all the same files, cascading the windows.

Searching for Text in Multiple Files

Inserting One File into Another

Dialog Box Settings

This command lets you copy an existing file into your current file.

To insert a file into your current file:

1. Position the cursor where you want to insert the file.
2. Choose Insert from the File menu to display the Insert File dialog box.
3. Enter the name of the file you want to insert in the File Name text box.
   You can use the Drives, Directories, and List Files of Type list boxes to find the file you want to use.
4. Click OK.
   The ScriptMaker Editor inserts the entire file at the insertion point.

Moving and Copying Text

See Also

The Cut and Copy commands both let you move text to the Windows Clipboard and place it somewhere else in the same or another file. You can even move text into another text-oriented application, such as a word processor.

To move text from one place to another in the same file:

1. Select the text you want to move.
2. Choose Cut from the Edit menu.
   Or,
   Click the Cut button.
3. Move the cursor to the position you want the text to go.
4. Choose Paste from the Edit menu.
   Or,
   Click the Paste button.

To copy text from one place to another in the same file:

1. Select the text you want to copy.
2. Choose Copy from the Edit menu.
3. Move the cursor to the position you want the text to go.
4. Choose Paste from the Edit menu.
   Or,
   Click the Paste button.

To move (or copy) text from one file to another file:

1. Select the text you want to move or copy.
2. Choose Cut (or Copy) from the Edit menu.
   Or,
   Click the Cut button. (There is no Copy button.)
3. Open the other file. If it is already open, activate its file window.
4. Move the cursor to the position in the new file where you want the text to go.
5. Choose Paste from the Edit menu.
   Or,
   Click the Paste button.

You can set the ScriptMaker Editor so that it selects the current line as the Cut or Copy text automatically. See Setting Editor Preferences.

Moving to a Specific Line

Dialog Box Settings

You can move the cursor to a specific line or extend the selected text to a specific line using the Goto Line dialog box.

To go to a specific line:

1. Type the line number in the Line Number text box.
2. Click OK.

Comparing Two Files

See Also Dialog Box Settings

Perhaps you have saved information under confusing filenames, such as TEST1 and TEST2. Or you have two different versions of your program source code under similar names. The Compare command lets you display both versions for comparison.

To compare two files:

1. Choose Compare from the File menu to display the Compare dialog box.
2. Enter the names of the files you want to compare in the File 1 and File 2 text boxes.
   Or,
   Click the File 1 and File 2 drop-down list boxes to choose from a list of filenames you have compared before.
   Or,
   Click the Browse button to search drives, directories, and files.
3. Enter 1 in both Line text boxes to start from the beginning of each file.
   If you only want to compare part of the file, enter the line numbers where the comparison should begin (it may be different in each file).
4. Click the option button for Horizontal or Vertical display.
   Horizontal displays the files one above the other; Vertical displays them side by side.
5. Click OK.
   The ScriptMaker Editor displays both files, highlighting the first difference it finds. (If there is no match, ScriptMaker Editor displays "No Match Found".)
6. Click Next Match to go on.
   Or,
   Click Cancel to stop the comparison.

Using Multiple Edit Windows

Printing All or Part of a File

Dialog Box Settings

ScriptMaker Editor provides printing capabilities that should be sufficient for most of your needs, including font selection and page headers and footers. For more complex formatting, you will probably want to open your file in a more powerful word processor.

To print an entire file:

1. Make sure the file you want to print is active.
2. Deselect any selected (highlighted) text.
3. Choose Print from the File menu to send the file to the printer.

To print a portion of a file:

1. Make sure the file is active.
2. Select (highlight) the portion of the file you want to print.
3. Choose Print from the File menu to send the selection to the printer.

   TIP: Any headers or footers you have selected appear with the partial printout. You may want to modify either one to indicate that this is a partial file.

Using Multiple Edit Windows

See Also

The commands in the Window menu lets you look at all your current files at the same time, and provide standard ways to display and interact with them. In general, you should keep the application window maximized.

To see all current files at once:

- Choose Tile from the Window menu. You can cut, copy, and move text from one file to another while looking at both files at the same time.

NOTE: Tile is most useful when you have only a few open files. The more files you have open, the less of each file you can see.

To display the title bars of all your windows:

- Choose Cascade from the Window menu.

To enlarge a window to see more text:

- Double-click the title bar of the window you want to enlarge, or click its maximize button.

To activate a cascaded window:

- Click any part of the window to bring it to the front as the active window. Double-click the title bar of any window to maximize it.

To see two windows containing the same file at the same time:

1. Click a file window.
2. Choose New Window from the Window menu.
   You can scroll each window separately, although only one is active at any time.

To iconize a window:

- Click the minimize button to turn the window into an icon. Icons are displayed at the bottom of the application window.

To tidy up icons:

- If the application space becomes cluttered, choose Arrange Icons from the Window menu.

Using ScriptMaker Editor Macros

ScriptMaker Editor lets you record keyboard macros to automate simple, repetitive tasks inside the Editor. Macros are not saved between sessions, and you can only record and use one at a time. Use the Recorder utility in the Tools menu for more sophisticated macro handling.

To record a macro:

1. Choose Record Macro from the Edit menu.
   The status line displays "REC" to indicate that it is recording.
2. Enter the text, menu commands, keystroke functions, and so on that make up your automated task. Every keystroke you enter will be included in the macro. You can record up to 256 editing events.

   NOTE: You cannot record events that involve filling in dialog boxes.
3. Choose Stop Recording Macro from the Edit menu.
   The status line displays "Keyboard macro defined".

To play back the macro:

1. Position the cursor where you want the macro sequence to begin.
2. Choose Playback Macro from the Edit menu.

Saving a File with a New Name

See Also Dialog Box Settings

You may want to keep an older version of a file, such as an AUTOEXEC.BAT file, for reference. The best way to do this is to save it under another name using the Save As command. You can also save the file on a different drive and directory.

To save a file under another name:

1. Choose Save As from the File menu to display the Save As dialog box.
2. Enter a new name in the File Name text box.
3. If you want to save this file in a different directory, use the Drives and Directories list boxes to change it.
4. Click OK.

Saving Text as a Separate File

Saving Text as a Separate File

See Also Dialog Box Settings

The Write Block command lets you save a portion of a file as a new file. You select the text and assign it a new filename.

To write a block of text to disk:

1. Select (highlight) the text you want to save separately.
2. Choose Write Block from the File menu to display the Write Block dialog box.
3. Enter a name for the file in the File Name text box.
   You can use the Drives and Directories list boxes to specify where to save the new file.
4. Click OK.

Saving a File with a New Name

Choosing an Icon

Dialog Box Settings

When you make a script into an executable file with the extension .EXE, you can choose an icon that will display when the script runs or is placed on a desktop.

To choose an icon:

1. If you want an icon to display when the file is on the desktop or minimized, click Icon in the Save Exe As dialog box.
   The Choose Icon dialog box appears.
2. Type the name of a file that has an icon you would like to use in the Alternate Icon File drop-down combination box or click Browse... and use the Choose Icon From... dialog box to locate a file. As you make your selection, the icon (or icons) appear in the Icon box.
   Click Default to restore the default icon to the Icons box.
3. If more than one icon appears in the Icons box, click the icon of your choice.
4. Click OK to return to the Save Exe As dialog box.

NOTE: Most applications that execute in Windows have an icon you can "borrow." You cannot use a bitmap file as you would for Windows wallpaper.

Browsing for an Icon

Dialog Box Settings

You can examine the icons from any number of applications.

To see the icon associated with a file:

1. Select Icons from the List Files of Type drop-down list box.
2. Change drives and directories if you want to.
3. Select a file from the Files list box.
4. Click OK.
   The icon or icons for the selected file appears in the Icons box in the Choose Icon dialog box.

ScriptMaker Programming Tools

See Also

ScriptMaker not only is a full-fledged programming editor, but it also boasts a suite of tools used to help you create robust Windows and DOS scripts.

Recorder

Use the Recorder to record a series of events generated by the user within the Windows environment, then translate the recorded series of events into ScriptMaker statements. The statements can be included in a program or subroutine that can reproduce the series of recorded events.

Dialog Editor

Use the Dialog Editor to create your own dialog boxes that can be incorporated into ScriptMaker programs. You can start with an empty Dialog Editor window and build a custom dialog box piece by piece; or you can "capture" an existing dialog box from another Windows application into the Dialog Editor main window and then modify that dialog box to suit your purposes.

Reference dialog box

Use the Reference dialog box as a handy online method of quickly determining what the syntax is of each ScriptMaker statement and function. The Reference dialog box also provides a short description of the statement.

The ScriptMaker Language

See Also

This section introduces the basic components of the ScriptMaker language and provides a framework for programming concepts discussed in the language reference and procedures.

Using the Recorder

See Also

ScriptMaker's Recorder allows you to create a macro that perform actions outside the Editor and insert that macro into a script. Each macro ends with the QueFlush statement that causes the rest of the statements in the macro to be executed.

When creating a macro for a script, it is better to use keystrokes than mouse movements because dialog boxes and windows may be positioned differently when the macro is replayed and cause unexpected behavior from the macro.

To record a macro:

1. Choose Recorder from the Tools menu.
   The Insert Macro dialog box appears.
2. Set the recording options in the dialog box.
3. Click OK. You are now recording your macro. The Recorder window appears and remains active until you stop recording.
4. Use the keystrokes and mouse actions you want to incorporate into your program. You can pause at any time by clicking the Pause button in the Recorder window. Resume the session by clicking Pause again.
5. Click the End button in the Recorder window when you are finished recording.
   The Stop Recorder dialog box appears.
6. Select the option button that indicates where you want to insert the ScriptMaker macro statements.
7. Click OK.

Although you could type in the ScriptMaker statements corresponding to a macro, it is much simpler to use the Recorder to record the macro and generate the corresponding statements. You can use the statements in the macro as a way to make general adjustments to an application window, or you can use the statements as a skeleton into which you add statements that control dialog boxes and their components.

Statements Generated by the Recorder

Statements Generated by the Recorder

See Also

The Recorder generates statements, but you can modify those statements and add additional ones that you write yourself.

The table below lists the statements and functions associated with each of the possible types of actions that can be recorded. The syntax for each statement appears to the right of the statement.

ActivateControl
AppMaximize
AppMinimize
AppMove
AppRestore
AppSize
DoKeys
HLine
HPage
HScroll
Menu
QueEmpty
QueFlush
QueKeyDn
QueKeys
QueKeyUp
QueMouseClick
QueMouseDblClk
QueMouseDblDn
QueMouseDn
QueMouseMove
QueMouseUp
QueSetRelativeWindow
SelectButton
SelectComboBoxItem
SelectListBoxItem
SendKeys
SetCheckBox
SetEditText
SetOption
VLine
VPage
VScroll
WinActivate

The ScriptMaker Language
Recorder Statements by Function

Recorder Statements by Function

See Also

The list below groups the statements generated by the Recorder by their function.

Making an Application Active

See Also

Making an application active during a recording session, generates a WinActivate statement. The window is specified in the **WinActivate statement as a string expression that contains the title that appears in the window's** title bar.

For example, to make Norton Desktop the active window, use the following statement:

```
WinActivate "Norton Desktop"
```

Scrolling Statements

See Also

Moving the scroll box in a scroll bar generates:

- A line-scrolling statement (caused by clicking the arrow at either end of a scroll bar).
- A page-scrolling statement (caused by clicking the scroll bar at either side of the scroll box).
- A scroll box position statement (caused by dragging the scroll box to a new position within the scroll bar).

The Recorder only records statements when the scroll box moves in the active window. However, you can add statement manually for active dialog-box components.

Because scroll bars can be horizontal or vertical, the statements for line- and page-scrolling start with an H or a V. They are HLine, VLine, HPage, and VPage. Because the scroll bar movement can be in either of two directions, the value specified by the statement is either a negative or positive number of lines or pages. Scrolling to the left for HLine or HPage and up for VLine or VPage are indicated by negative numbers. A positive number indicates scrolling in the opposite direction.

Examples:

To scroll to the right 10 lines using the horizontal scroll bar:

```
HLine 10
```

To scroll to the left two pages using the horizontal scroll bar:

```
HPage -2
```

To scroll up one line using the vertical scroll bar:

```
VLine -1
```

To scroll down one page using the vertical scroll bar:

```
VPage 1
```

The HScroll and VScroll statements position the scroll box a percentage of the way down or across the total range of the scroll bar. The percentage is an integer.

Examples:

To set the horizontal scroll box in the middle of the scroll bar:

```
HScroll 50
```

To set the vertical scroll box at the very end of the scroll bar:

```
VScroll 100
```

Mouse and Keyboard Activity

See Also

The event queue stores the statements that specify mouse movements and keyboard selections as they are recorded. What statements end up in the queue as a result of the recorded movements and selections are explained in Adding Mouse Events to the Queue, Adding Keystrokes to the Queue and Sending Keystrokes Directly to an Application.

The QueFlush statement plays those statements, which empties the queue. It appears in the macro after each series of keystrokes and mouse movements. For example, if you activate a new application window, QueFlush is inserted before the WinActivate statement so the keystrokes and mouse movements are replayed in the application to which they apply.

QueFlush's parameter, when TRUE, indicates that the state of CAPSLOCK, NUMLOCK, SCROLL LOCK, and INSERT as they were prior to playing the statements in the event queue should be restored after QueFlush is complete. When FALSE, those keys are in the state they were left in by the statements in the event queue. QueFlush is complete only after all the events in the queue have been played. The QueEmpty statement, which takes no parameters, can be used to explicitly empty the queue without first having to play the events contained in the queue. Using a QueFlush statement immediately after a QueEmpty statement plays no events, because the queue is already empty.

Examples:

```
QueFlush TRUE     'Play back events in the queue and save
                  '  states
                  'Assuming the queue contains some events
QueEmpty          'Empty the queue without first playing
                  '  the events
QueFlush TRUE     'No events are played since the queue
                  '  has been emptied
```

NOTE: The Recorder does not generate the **QueEmpty** statement. You can insert it into the code if it is needed.

Adding Mouse Events to the Queue

See Also Examples

Move movement events are added to the queue when you move the mouse.

All mouse movements are relative either to a window or to the screen. The QueSetRelativeWindow statement sets all subsequent mouse movements relative to a specified window. The next QueFlush statement will use the new setting for playing mouse events stored in the event queue. After a QueFlush statement, mouse events are reset to be relative to the screen, unless another QueSetRelativeWindow is executed to give a new setting. The window is specified along with its handle in the Windows environment. Using a handle with the value 0 sets mouse movements to be relative to the active window. The Recorder generates the QueSetRelativeWindow if the mouse relative to option was set to be the active window.

The QueMouseMove statement adds a mouse movement to the event queue that indicates a new position for the mouse pointer. It requires an x-coordinate and a y-coordinate (both integers) for the new location of the mouse pointer in pixels. The Recorder generates QueMouseMove's when the mouse is being dragged.

Each of the next five statements for entering a mouse event into the queue uses the following three parameters:

button	The button used for generating the event. VK_LBUTTON is the constant for indicating the left button; VK_RBUTTON is the constant for indicating the right button.
x	The x-coordinate (in pixels) of the mouse when the event occurred.
y	The y-coordinate (in pixels) of the mouse when the event occurred.

QueMouseClick: Adds a single-click event, which consists of a mouse button pressed down and immediately released.

QueMouseDblClk: Adds a double-click event, which consists of a single-click immediately followed by another single-click.

QueMouseDblDn: Adds a mouse double-down event, which consists of a mouse button pressed down and released followed by the mouse button pressed down.

QueMouseDn: Adds a mouse button down event.

QueMouseUp: Adds a mouse button up event.

Examples

QueMouseMove Example:

```
QueMouseMove 100, 100
'Mouse pointer moves to x = 100 and y = 100
```

QueSetRelativeWindow Example:

```
'Adjust mouse coordinates relative to Notepad
'Get the handle to the Notepad window
hWnd = WinFind("Notepad")
QueSetRelativeWindow hWnd
```

QueMouseClick Example:

```
'Left mouse button click at (x=167, y=205)
QueMouseClick VK_LBUTTON, 167, 205
'Play the click
QueFlush TRUE
```

QueMouseDblClk Example:

```
'Right mouse double click at (x=100, y=101)
QueMouseDblClk VK_RBUTTON, 100, 101
'Play the double click
QueFlush TRUE
```

QueMouseDblDn Example:

```
'Left mouse button double down (x=89, y=149)
QueMouseDblDn VK_LBUTTON, 89, 149
```

QueMouseUp Example:

```
'Left mouse button up (x=100, y=149)
QueMouseUp VK_LBUTTON, 100, 149
'Play the double down and up
QueFlush TRUE
```

QueMouseDn Example:

```
'Right mouse button down (x=171, y=137)
QueMouseDn VK_RBUTTON, 171, 137
'Right mouse button up (x=171, y=137)
QueMouseUp VK_RBUTTON, 171, 137
'Play the down and up
QueFlush TRUE
```

Adding Keyboard Events to the Queue

See Also Examples

The Recorder can generate statements corresponding to two slightly different types of keyboard events. The first type consists of keystroke events that are stored in the event queue and then later played using QueFlush. These keystrokes are placed in the event queue with the QueKeys, QueKeyDn, and QueKeyUp statements. QueKeys places full keystrokes (key down followed by key up) into the queue, while QueKeyDn and QueKeyUp indicate that a key was held down for some time before it was released. Each takes a string parameter containing the keystrokes to add to the event queue.

The second type of event consists of keystroke events that occur without first being stored in the event queue. This is done with the DoKeys statements, which also take a string parameter of keystrokes. Instead of adding the keystrokes to the event queue, it plays them immediately. When no mouse events or partial keystrokes make an event queue necessary, ScriptMaker optimizes the recored keystrokes using DoKeys.

QueKeys, QueKeyUp, QueKeyDn, and DoKeys all use the same format for specifying the keystrokes in the string parameter:

To specify any printable character from the keyboard:

- Use that key (for example, "h" for lowercase h, and "H" for uppercase h).

To specify a sequence of keystrokes:

- Append keystrokes, one after the other, in the order you want (for example, "asdf" or "dir /p").

To specify special characters:

- The plus sign (+), caret (^), tilde (~), percent sign (%), parentheses, square brackets, and curly braces specify keystroke combinations. For example "^d" indicates Ctrl+D. These special uses appear later in this list. To specify one of these characters as itself, a single or shifted keystroke with no special meaning, enclose the corresponding character within curly braces. For example, "{(}" specifies a left parenthesis, or "{%}" specifieies the percent symbol).

To specify keys that are not displayable characters:

- Enclose the description of the key within curly braces. For example, {ENTER} is the Enter key and {UP} is the UpArrow key. Click here  to see a list of these keys.

To specify keystrokes combined with a modifier key, such as Shift, Ctrl, or Alt:

- Precede the keystroke specification with "+", "^", or "%" respectively. For example, `"+{ENTER}"` means Shift+Enter, `"^c"` means Ctrl+C, `"%{F2}"` means Alt+F2).

To specify a modifier key combined with a sequence of consecutive keys:

- Group the key sequence within parentheses and precede it with either "+", "^", or "%" (for example, "+{abc}" means that you would replicate the sequence by holding the Shift key down while typing the a, b, and c keys in consecutive order, "^({F1}{F2})" means the Ctrl key is held down while the F1 and then the F2 keystrokes are given).

To embed the ENTER keystroke within a key sequence:

- The "~" is a shortcut for embedding the ENTER keystroke within a key sequence. For example, "ab~de" means the Enter key was pressed after "ab".

To embed quotes:

- Use two quotes in a row, for example, "This is a ""test"" of the system".

To repeat a keystroke:

- Enclose the keystroke and a repeat count within curly braces; for example, "{a 10}" means "Produce 10 "a" keystrokes"; "{ENTER 2}" means "Produce two ENTER keystrokes".

Examples

QueKeys Example 1:

In this example, the Recorder records the NUMLOCK keystroke followed by the 1, 2, and then 3 keys typed on the numeric keypad, and finally the Enter key. The QueFlush statement then plays all five of the keystroke events:

```
QueKeys "{NUMLOCK}{NUMPAD1}{NUMPAD2}{NUMPAD3}{ENTER}"
QueFlush TRUE
```

QueKeys Example 2:

In this example, also generated by the Recorder, the "a" key has been combined with the Ctrl key (indicated by the "^") , the Alt key (indicated by the "%"), and the Shift key (indicated by the capitalized "A"):

```
QueKeys "^(%(A))"    [[***Why doesn't ^%A work? or does it?]]
QueFlush TRUE
```

QueKeys Example 3:

The following example shows the sequence of recorded events: the Norton Desktop window becomes active and Ctrl+D keystroke (the hot key for running DOS) is pressed. While in DOS, the Recorder is not active. After you type `EXIT` and press the Enter key, Windows is reactivated and so is the Recorder, which records the key up event of the Enter keystroke:

```
WinActivate "Norton Desktop"
QueKeys "^(d)"                  'Go to DOS
QueKeyUp "{ENTER}"              'Coming back to Windows
QueFlush TRUE
```

NOTE: The Recorder cannot record events that occur in DOS, as the previous example shows. In addition, no events, including keystrokes, can be played while in DOS. This means that the above example, if replayed, would not return to Windows from DOS even if `QueKeys "exit{ENTER}"` was added.

QueKeys Example 4:

Keyboard activity that modifies mouse events is also recorded. In this example, the Shift key is held down, the mouse dragged, and the Shift key released:

```
'Hold down the Shift key
QueKeyDn "{+}"
'Press the left button and start dragging
QueMouseDn VK_LBUTTON, 204, 103
'Release the mouse button
QueMouseUp VK_LBUTTON, 443, 350
'Release the Shift key
QueKeyUp "{+}"
QueFlush TRUE
```

Non-Displayable Characters

{BACKSPACE}	{BS}	{BREAK}	{CAPSLOCK}
{CLEAR}	{DELETE}	{DEL}	{DOWN}
{END}	{ENTER}	{ESCAPE}	{ESC}
{HELP}	{HOME}	{INSERT}	{LEFT}
{NUMLOCK}	{NUMPAD0}	{NUMPAD1}	{NUMPAD2}
{NUMPAD3}	{NUMPAD4}	{NUMPAD5}	{NUMPAD6}
{NUMPAD7}	{NUMPAD8}	{NUMPAD9}	{NUMPAD/}
{NUMPAD*}	{NUMPAD-}	{NUMPAD+}	{NUMPAD.}
{PGDN}	{PGUP}	{PRTSC}	{RIGHT}
{TAB}	{UP}	{F1}	{SCROLLLOCK}
{F2}	{F3}	{F4}	{F5}
{F6}	{F7}	{F8}	{F9}
{F10}	{F11}	{F12}	{F13}
{F14}	{F15}	{F16}	

Sending Keystrokes Directly to an Application

See Also Examples

The SendKeys statement, which the Recorder does not generate, sends keys to the active application directly. You specify the keystrokes for this statement using the same format as for the statements described above. The second parameter is optional; it can be either TRUE or FALSE, and is used only for compatibility with other BASICs. ScriptMaker always waits for the keystrokes to be completely processed before continuing to the statement after SendKeys.

Examples

SendKeys Examples:

```
  SendKeys "CONTINUE", FALSE       'Execution continues
                                   '  immediately
  SendKeys "WAIT"                  'Process all keys first
                                   '  before continuing
  SendKeys "WAIT", TRUE            'Also processes all
                                   '  keys first before
                                   '  continuing
```

Window Management Statements

See Also Examples

The Recorder can record high-level window management events such as moving, maximizing, minimizing, restoring, and sizing windows.

The AppMove statement moves a window to a given (x,y) pixel location. The statement has no effect if the window is currently in the maximized state. The parameters are integers. The third parameter specifies the window to move. The parameter is a string expression containing the title of the window. Moving the window does not change its state (active or inactive).

The AppMaximize statement maximizes a window and makes it active. You specify the window to maximize by providing its title. If no window is specified, the window that is currently active is maximized.

The AppMinimize statement minimizes a window, but the statement does not change its state. You can specify the window to minimize by providing its title. If no window is specified, the window that is currently active is minimized.

The AppRestore statement restores a window if it is currently either minimized or maximized, but the statement does not change its state. You specify the window to restore by providing its title. If no window is specified, the window that is currently active is restored. If the specified window is currently in the restored state, nothing happens.

The AppSize statement sets a window to a given width and height in pixels, but does not change its state. The statement has no effect if the window is either currently maximized or minimized. The first two parameters, the new width and height, are integers. The third parameter specifies the window to resize. The parameter is a string expression containing the title of the window.

Examples

Example:

```
AppMaximize "Notepad"       'Maximizes Notepad and
                            '  makes it active
AppMinimize                 'Minimizes Notepad
AppRestore                  'Restores Notepad
AppSize 400, 300            'Makes the Notepad
                            '  window 400x300 pixels
                            '  in size
AppMove 0, 0                'Moves the window to the
                            '  upper-left corner
```

Menu Statements

See Also Examples

Interactions with an application's menus generate the Menu statement. When you select a menu item, the Menu statement that is generated has a string expression containing the name of the selected menu. The complete menu item name is given, with each menu level separated by a period. For example, "File.Open" specifies the Open command in the File menu.

You can specify a menu item (or menu command) by using its numeric position within the menu. For example, `"#3.#4"` selects the fourth item from the third menu.

You can select items from the active window's Control menu by beginning the menu item specification with a period. For example, ".Restore" specifies the Restore command in the Control menu.

Menu Examples

Examples:

```
'Select the Exit item from the File menu
Menu "File.Exit"
'Select Bold from the third level
Menu "Format.Character.Bold"
'Select the Maximize item from the system menu
Menu ".Maximize"
'Select the second item from the File menu
Menu "File.#2"
```

Dialog Box Statements

See Also Examples

This section describes six statements that apply to dialog-box components, called controls. The first parameter for each statement identifies the control to which it applies. A dialog control can be identified either by its name (a string expression) or its ID (an integer expression). For command buttons (also called push buttons), option buttons (also called radio buttons), and check boxes, the name is the text that appears on or is associated with the button. For list boxes, combination boxes, and text boxes, the name is the string of text immediately preceding the control.

The ActivateControl statement makes the specified control active.

The SelectButton statement simulates a mouse click on a button. The parameter identifies the button on which to simulate the click, and is specified using either the button's name or its ID.

The SelectComboBoxItem or SelectListBoxItem statement selects an item from a combination or list box, respectively. The first parameter identifies the box and the second identifies the item to select. The item is identified either by its name (a string expression), or its line number (an integer). A third optional parameter specifies whether the item is selected using a double-click or a single-click. If not specified or set to FALSE, the item is selected using a single-click. A TRUE value indicates a double click.

The SetCheckBox statement sets the state of a check box. The first parameter identifies the check box and the second gives its state as an integer. If the state is 1, the box is checked. If the state is 0, the check is removed. If the new state is 2, the box is grayed (only applicable for three-state check boxes).

The SetEditText statement sets the contents of a text box. The first parameter identifies the text box and the second supplies the contents of the box as a string expression.

The SetOption statement simulates a click on an option button. The first parameter identifies the option button on which to simulate the click.

Examples

ActivateControl Example:

To make a custom control active within a dialog box, first make a known control immediately preceding the custom control active, and then simulate a Tab keystroke:

```
ActivateControl "Portrait"
DoKeys "{TAB}"
```

SelectButton Example:

```
SelectButton "Cancel"       'Click on the Cancel
                            '  button
```

SelectComboBoxItem Example:

The following is a simple recorded example where an application is activated and an item is selected from a combination box:

```
'Make the SuperFind application active
WinActivate "SuperFind"
'Select the [All Drives] item in the "Where:" combination box
SelectComboBoxItem "Where:","[All Drives]"
```

SelectListBoxItem Example:

The following is a simple recorded example in which an application is activated, and an item is selected from a list box:

```
'Select the Box Types application
'Make the Box Types application active
WinActivate "Box Types"
'Select the "Big" item in the "Box Size" list box
SelectListBoxItem "Box Size", "Big"
```

SetCheckBox Example:

```
WinActivate "Control Panel|Desktop"
'Remove the check
SetCheckBox "Wrap Title", 0
```

SetEditText Example:

```
WinActivate "Control Panel|Desktop"
'Set the text in the "Delay:" edit control to "12"
SetEditText "Delay:", "12"
```

SetOption Example:

```
WinActivate "Control Panel|Desktop"
'Click on the "Center" option button
SetOption "Center"
```

The Reference dialog box

The Reference dialog box provides a quick alphabetical reference to all the ScriptMaker statements. Use it to double-check the syntax of a particular statement or function.

Each statement and function appears in the list box at the left, along with a description and syntax summary of each statement.

You can keep the Reference box open as you create or edit the script file. Use Alt+F6 to toggle back and forth between the Reference box and the text-editing screen.

To add the beginning of a statement or function to your script:

- Double-click the statement or function's name in the Command list box.

Contacting Technical Support and Customer Service

To quickly find technical support or customer service information, click on one of the following:

Customer Service, U.S. and Canada

Technical Support, U.S. and Canada

Symantec BBS

Fax Retrieval System

Customer Service and Technical Support, International

Customer Service (United States and Canada only)

Symantec Corp.
175 W. Broadway
Eugene, OR 97401

(800) 441-7234 voice
(503) 334-7474 fax
Hours: 7:00 A.M. to 5:00 P.M. Pacific Time
Monday through Friday

Technical Support (United States and Canada only)

Symantec Corp.
175 W. Broadway
Eugene, OR 97401

(503) 465-8420 for Norton Desktop.
(503) 465-8450 for Norton AntiVirus and Norton Backup
(503) 334-7470 fax
Hours: 7:00 A.M. to 5:00 P.M. Pacific Time
Monday through Friday

Symantec BBS

300-, 1200-, and 2400-baud modems	(503) 484-6699 (24 hrs.)
9600-baud modems	(503) 484-6669 (24 hrs.)

Settings for the Symantec BBS are:

- 8 data bits, 1 stop bit; no parity

Fax Retrieval System (United States and Canada only)

Symantec's Fax Retrieval System provides instant access to general product information, technical notes and virus definitions through a 24 hour automated attendant. To access this service, simply have your fax number ready and dial (800) 554-4403 from any fax machine or touch-tone phone.

International Technical Support and Customer Service

United Kingdom	Symantec UK Limited Sygnus Court Market Street Maidenhead Berkshire SL6 4AD United Kingdom	0628 59 222 voice 0628 592 287 fax
Europe (all countries except UK)	Symantec Europe Kanaalpark 145 Postbus 1143	31 71 353 111 voice 31 71 353 150 fax

	2321 JV Leiden The Netherlands	
Australia	Symantec Pty. Ltd. Upper Level 408 Victoria Road Gladesville, NSW 2111 Australia	61 2 879 6577 voice 61 2 879 6805 fax
All other countries	Symantec Corp. 10201 Torre Ave. Cupertino, CA 95014 U.S.A.	(408) 252-3570 voice (408) 253-4992 fax

Close

ScriptMaker Glossary

A

accelerator key
active
active window
allocate
alphanumeric
ANSI
append
array
array element
ASCII
ASCII file
assignment operator
assignment statement
asterisk
attribute

B

Backus-Naur form (BNF)
batch file
binary file
bitmap
Boolean expression
buffer
button bar

C

call
cascade
cascading menu
case sensitive
case construct
character
character code
check box
click
client/server
Clipboard
combination box
command
command button
comment

compare
compile
compiler
composite data type
concatenate
condition
conditional
conditional construct
conditional expression
constant
constant expression
control
control construct
Control menu
Control-menu box
Control Panel
construct
current directory
cursor

data declaration
data type
data validation
data value
declaration
default
delimit
delimiter
dialog box
dialog unit
dimmed
directory
display
document
document window
DOS
double
double-click
drag
drag and drop
drop-down combination box
drop-down list box
Dynamic Data Exchange
dynamic dimensioning



landscape
list box
literal
local variable
logical expression
logical operator
long

macro
mask
mathematical expression
menu
maximize
Maximize button
menu bar
menu item
menu-driven
memory
menu
menu bar
message
Microsoft Windows
minimize
Minimize button
mnemonic
modal dialog box
modeless dialog box
modulo
monospaced
mouse
mutual exclusion

nesting
NOT
null string
numeric expression

octal
operand
operator
operator precedence
option button
OR
output

parameter

parameter passing
parse
pass by reference
pass by value
password box
path
pathname
pixel
point
portrait
power
precedence
predefined function
predefined subroutine
primary mouse button
procedure
program
program file
programming language
prompt button
push button

radian
radio button
random number generation
range
read
record
Recorder
recursion
regular expression
relational expression
relational operator
replace
restore
reserved word
return
routine

scope
script
ScriptMaker
scroll arrow
scroll bar

scroll box
scrolling
search
search and replace
search string
secondary mouse button
seed
select
separator line
serial format
shell
show
sign
significant digits
simple data type
single
spin button
statement
status bar
string
string expression
string variable
subprogram
subroutine
subscript
substring
syntax error

text file
tile
title bar
toolbar
truncate
tutorial
twip
two-dimensional
two-dimensional array
type
type declarator

user-defined constant
user-defined dialog box
user-defined error
user-defined function
user-defined subroutine

value
variable
variable expression
version
viewport
Viewport

wallpaper
wildcard
WIN.INI
window
window corner
window handle
Windows
write

x-axis

y-axis

.EXE
In DOS, a filename extension that indicates an executable file. To run the program, type the filename (without the extension) at the command prompt and press Enter. In Windows, you can simply double-click the filename.

accelerator key
The underlined letter indicating that a menu, menu item, or control can be accessed using a combination of Alt plus the underlined letter. For example, most Windows applications display the File menu when you press Alt+F. Once a menu is displayed, pressing the key corresponding to an underlined letter in a menu item executes that menu item.

active
The window or icon that you are currently using or that is currently selected. Keystrokes and commands affect the active window. To differentiate the active window from other windows, its title bar changes color. To differentiate the active icon from other icons, its label changes color.

Windows or icons on the desktop that are not selected are *inactive*.

active window
The currently selected window, which always appears on top of any other window. The title bar in the active window is a different color or intensity than the title bar in an inactive window.

allocate
To set aside an amount of memory for a program's use.

alphanumeric
Made up of both letters and numbers, usually including spaces, special characters, and control characters. Ordinary text (such as this glossary entry) is alphanumeric.

ANSI
Acronym for the *American National Standards Institute*. This organization of American industry and business groups develop trade and communications standards. The standard 7-bit character has 256 possible values (ranging from 0 to 255). This value is the character's ANSI value. The values from 0 to 127 are known as the ASCII character set. Values from 128 to 255 are not part of the standard and are assigned different sets of characters by computer manufacturers and software developers.

append
To add to the end of something, such as adding data to the end of a file or text to the end of a document.

array
A composite data type. It has one variable name but any number of parts (called elements) that are all of the same simple type (integers, longs, singles, doubles, or strings). Each element is identified by the variable name and the *subscript* (or index), a number that indicates the elements position in the array. If an array has more than one dimension, each element has a subscript for each dimension. The subscripts are enclosed in parentheses and separated by commas. An array can have up to 60 dimensions.
Using an array in a script is a method of organizing or structuring data values or elements that are all the same type.

array element
A data value in an array. Each array element can (usually) be treated as a stand-alone variable. A specific element can be referenced by combining the array name with a set of subscripts.

ASCII

Acronym for *American Standard Code for Information Interchange*. Numeric values are assigned to letters, numbers, punctuation marks and a few additional characters. Computers and computer programs using these standard codes are able to exchange information.

Values 0 to 31 are assigned as control codes, such as backspace and carriage return, and are generally non-printing characters. In some fonts, these characters represent graphical symbols.
Values from 32 to 127 represent the numbers 0 to 9, common punctuation marks and the upper- and lowercase letters of the Roman alphabet.

ASCII file
A file containing characters or text without machine code or control characters. An ASCII file can contain carriage returns and spaces and an end of file marker, but is otherwise in a "generic" format. *See* text file.

assignment operator
An operator that assigns a value to a variable. The assignment operator in ScriptMaker is =.

assignment statement
In programming, a statement that assigns a value to a variable. An assignment statement usually has three parts: a variable, an operator, and an expression. When the expression is evaluated, the resulting value is stored in the destination variable. The ScriptMaker assignment operator is =.er is =.

asterisk
The character (*). In DOS, a wildcard that can be used in place of zero or more characters in a file search or with other commands. In ScriptMaker, the asterisk indicates multiplication.

attribute
One of the properties of an object. For example, a file on your local disk drives (floppy and hard disks) can have up to four attributes: archive, hidden, read-only, and system. In databases, the name or structure of a field is said to be one of its attributes. The attributes of your screen display control the color, underlining, blinking cursor, and so on.

Backus-Naur form (BNF)
A meta-language used to describe the syntax of other programming languages.

batch file
A batch file contain a series of commands executed in sequence by the computer instead of being entered one by one at the keyboard. Often used to automate routines as a short-cut for keystroke entry. DOS batch files have the file extension .BAT.

Boolean expression
A mathematical expression using operators such as AND, OR, and NOT to define logical conditions. Such an expression yields a Boolean value (TRUE or FALSE). *See* logical expression.

binary file
A file consisting of 8-bit data or executable code. A binary file is machine-readable only, often compressed or constructed so that only a particular program read it.

bitmap
A file containing a picture, stored as a set of colored dots or pixels (**pic**ture **el**ements), including information about its location and size. *See* pixel.

buffer
A portion of memory used to hold data temporarily until it can be transferred to its ultimate destination.

button bar
A row of buttons that let you perform specific tasks, such as copying, moving and deleting items or displaying certain information.

call
In programming, to transfer execution to some other part of a program, such as a subroutine.

cascade
Arranging open windows one upon the other, so that the upper and left screen bars are visible of the lower windows.

cascading menu
A menu that drops down from a menu item when that item is chosen.

case sensitive
Indicates that the case of the letters in a word can affect the meaning of the data. Not usually found in a DOS environment, although prevalent in the UNIX world. ScriptMaker is not case-sensitive.

case statement
A programming construct that requires a match of parameters or variables to determine which of several sets of instructions to execute. Used when evaluating a situation that can lead to different results.

character
A letter of the alphabet, a digit, or another computer symbol that can be used in a string. Each character is one byte long.

character code
The computer code that represents a particular character in a character set. For example, the ASCII code for the cent sign (¢) is 0162.

check box

A dialog box component that acts like a switch, representing an option that you can turn on or off. Some check boxes may have more than two options; each click cycles through the check boxes' options.

click
To press and release the primary mouse button.

client/server
A system in which the client requests information or services from the server and the server responds.

Clipboard
A buffer area in memory where data is stored when being transported from one Windows application to another.

combination box
A dialog box component that combines the capabilities of a text box and a list box. As in a text box, you can enter information into the entry field; like a list box, it provides a list of choices.

command
An instruction that causes a certain action to be carried out. You usually enter commands at the keyboard, from a menu, in a batch routine, or with an alternate input device such as a mouse.

command button (push button)
A rectangular button that carries out the action described by the text on the button. The two most common command buttons are OK (acknowledges a warning or message, or performs an action) and Cancel (closes a dialog box without performing any pending action).

comment
Also called *remark*. Text embedded in a program, usually describing what the program does, who wrote it, how and why it was corrected or changed, and so on. Because the compiler ignores anything marked as a comment ("commented out"), comments serve to document a program internally.

compare
To check two items, such as files or values, to determine whether they are alike or different.

compile
To translate all the source code of a program into executable code. A program that performs this task is called a *compiler*.

compiler
A program that translates the source code of a program so it can be executed.

composite data type
A data type that is composed of more than one part. For example, an array can have several parts called elements and is a composite data type.

concatenate
To combine two data strings so that one immediately follows the other.

condition
The state of an expression or a variablefor example, a result can be either true or false.

conditional
An action or operation that takes place based on whether or not a certain condition is true.

conditional construct
A construct containing several possible sequence of statements, only one of which is executed. Which one is executed depends on matching an expression or finding a logical expression that is true. The conditional constructs are the **If** and **Select Case** constructs.

conditional expression
An expression that yields a value of true or false, sometimes called a logical expression or Boolean expression. *See* logical expression.

constant
A named item that retains a constant value throughout the execution of a program, as opposed to a variable, which can have its value changed during execution. *See* variable.

constant expression
An expression composed only of constants; hence, an expression whose value does not change during program execution.

construct
Statements that are used together as a syntactical unit such as If...Then...Else and Do...Loop.

control construct
A construct that affects the flow of execution through a program. Control constructs include conditional constructs, loops, and go-to-label constructs.

control
A component of a dialog box that can be selected. For example a text box is a control.

Control menu
A menu that allows you to manipulate a window, dialog box or icon, or switch to another application or document. The Control menu is revealed when you click the Control-menu box. Sometimes called the *System menu*.

Control-menu box

Located in the upper-left corner of each window or dialog box. When you click this box, the Control menu drops down. Sometimes called the *System box*.

Control Panel
A Windows application that allows you to modify the Windows environment, such as adding printers and fonts, or adjusting the tracking speed of your mouse.

current directory
The directory that you are currently working in.

cursor
A flashing line, point, square or dot that indicates the position of data entry.

data declaration
A statement within a program that specifies the characteristics of a variable. Most programming languages allow (or require) you to specify a variables name and data type and possibly its initial value as well.

data type
In programming, a definition of a set of data that specifies the possible range of values of the set, the operations that can be performed on the values, and the way in which the values are stored in memory.

data validation
The process of testing to make sure data is current, consistent, within established boundaries, and otherwise accurate.

data value
The literal or interpreted meaning of a data item (such as an entry in a database field) or a data type (such as an integer) that is used as a variable.

declaration
The binding of an identifier to the information that relates to it. For example, to "declare a constant" means to bind the name of the constant with its value. Declaration usually occurs in the source code of a program; the actual binding can take place at compile time or at runtime.

declarator
The symbol used to declare the type of a variable. When a declarator is used, it is added as the last character of the variable's name the first time it is used in a routine.

default
A preset choice used by a script as the value of a parameter unless you explicitly specify another choice.

delimit
To set the limits of some entity using a special character called a delimiter. ScriptMaker delimits strings, comments, and so forth.

delimiter
A special character that separates individual entities or marks their endpoints. For example, commas separate or delimit parameters in a function or subroutine call. Double quotation marks indicate the endpoints of string literals. A single quotation mark separates a comment from a statement, and carriage return/linefeeds separate statements in a script.

dialog box
A rectangular area on that screen that either requests or provides information. Many dialog boxes ask for information Norton Desktop needs before it can complete a command. Other dialog boxes display warnings and other system messages.

dialog unit
A unit of measure commonly used in Windows dialog boxes because it is relatively independent of the resolution of the monitor or other display device. A horizontal dialog units based on the average character width of the font divided by 4; a vertical dialog unit is based on the character height of the font divided by 8. For example, if a component's size is 40 horizontal dialog units and 16 vertical units, that component is 10 characters wide and 2 characters high. For example, Helvetica characters are nearly twice as high as they are wide, making horizontal and vertical dialog units roughly the same size.

dimmed
An unavailable menu item. A dimmed item cannot be accessed and appears in light gray.

directory
A way of grouping files together on a disk. The root directory contains files and other directories, called subdirectories.

display
The visual output of a computer, as it appears on a video screen; commonly a CRT-based video display.

document
A file that is created by or associated with an application.

document window
A window that displays an application document, such as a spreadsheet or text file.

DOS
Disk **O**perating **S**ystem. Not exclusive to personal computers. Also shorthand for Microsoft DOS, or MS-DOS, the most common operating system for PCs. DOS manages the computer systems resources, such as memory, disks, keyboard, files, and so forth, and provides an interface between the user and application programs.

double
A simple numeric data type consisting of numbers in the range +/-1.7E+/-308. The size of a double-precision number is 8 bytes (64 bits: 1 for sign, 11 for the exponent, and 52 for the mantissa). The declarator for this type is the pound or number sign (#). A double-precision number has 15-16 significant digits.

double-click
To press the primary mouse button twice, in rapid succession. Generally used to select an item.

drag
To hold the primary mouse button down while moving the mouse in a given direction. For example, you can use this technique to choose a menu item, move or resize a window, or as part of a drag-and-drop operation.

drag-and-drop
A feature that lets you use the mouse to perform complex operations with instant visual feedback. This feature involves dragging a file, directory or icon from one location and "dropping" it (by releasing the mouse button) on another location.

drop-down combination box
A special type of combination box that reveals a list of choices when you click its prompt button.

drop-down list box
A special type of list box that reveals a list of choices when you click its prompt button.

Dynamic Data Exchange (DDE)
A form of interprocess communication (IPC) implemented in Microsoft Windows and OS/2. When two or more programs that support DDE are running simultaneously, they can exchange information and commands.

dynamic dimensioning
Assigning the space for each dimension at runtime rather than at compile time. A **ReDim** statement can change an array at runtime.

editor
A simple program that allows you to create and modify text files. Similar to a word processor, but usually less powerful.

empty string
A string containing no characters. An empty string has a location in memory, but there is nothing stored at that location. In a script an empty string is represented as an empty string literal (""). This is not the same as a null string.

end-of-file (EOF)
The code that a program places after the last byte of a file to mark that there is no more program data.

entry
An item of information treated as a unit. For example, each line in a section of an .INI file is an entry.

environment
The resources available to the user in a particular computer system. For examples, Microsoft Windows is known as a windowing environment.

environment variable
A variable used to store a piece of information needed by the operating system, such as the directories in your path, the location of the command processor (typically COMMAND.COM) and what to put in your DOS prompt. Environment variables are generally set up in the AUTOEXEC.BAT and CONFIG.SYS files.

error message
A message from the system or a program advising the user of an error that requires human intervention in order to be solved.

event queue
A list of mouse and keyboard events recorded as a macro by the Recorder and inserted as statements into a script. The statements are executed by the **QueFlush** statement.

Exclusive OR
XOR. A logical operation that yields TRUE if and only if one of its operands is true and the other is false.

executable file
A file containing a program that is in the proper format and ready for DOS or Windows to run.

execute
To load a compiled program into memory and run it.

expression
A combination of symbols (identifiers, values, and operators) that yields a result when evaluated. In programming, the resulting value might then be assigned to a variable, passed as an parameter, tested within a control construct, or used as part of another expression.

extension
The one, two or three letters after the period in a filename. In the filename AUTOEXEC.BAT, the extension is .BAT. An extension often identifies the type of file; for example, .EXE identifies an executable file, whereas .DOC is a common file extension for files created by word processors.

field
1) The memory location that stores the value of a dialog box control. The fields name is used to preset or retrieve data from the dialog box control. 2) A column in a database or a data item in a record.

file attribute
A setting for a file that indicates a trait of the file. File attributes can also restrict a file's use.

file extension
In a filename, the 3-letter suffix that follows the period; for example, the .BAT in AUTOEXEC.BAT is the file extension.

file pointer
Keeps track of the next position to be read from or written to a file. The first position in the file is position 0.

file specification
Determines a file or set of files that is the target of some operation, such as copy, erase, or find. A file specification may include DOS wildcard characters, as in *.EXE, or ????90.DOC.

filename
The name of a file. See pathname.

find
See search.

function
A group of statements that must be declared as a unit and performs some task. A function is executed when its name is used in (or called from) another function or subroutine. Its name is used as part of an expression and returns a value. Parameters are passed between the function and the routine that called it.

global
Universal, in the sense of being related to an entire file, document, script, or other entity. A global variable in a script is one that can be accessed by modules (routines) other than the one in which the variable is defined. ScriptMaker has no global variables. *See* global operation.

global operation
An operation that affects an file, document, or program. A global search-and-replace operation, for example, finds one word and replaces it with another throughout a document.

global variable
A variable whose value can be accessed and modified by any statement in a program. That is, the variable is available to the entire program, including statements and functions. ScriptMaker has no global variables.

GOTO statement
A control statement used in programs to transfer execution to some other statement. The high-level equivalent of a branch or jump instruction.

group box
A dialog box component that organizes related choices. A group box consists of a label and a border and often contains check boxes and/or option buttons.

handle
A unique number used to identify a device or an object such as a file or window in a graphical interface.

help
A disk-based form of assistance, also called online help, provided by many application programs, consisting of advice or instructions on using program features. Help can be accessed directly by pressing F1, without interrupting work in progress or searching through a manual.

hexadecimal
The base-16 number system that consists of the digits 0-9 and the letters A through F. Commonly used in programming to represent the binary numbers used by the computer.

hide
To remove an application from visibility.

identifier
Generally, any text string used as a label, such as the name of a subroutine or a variable in a script.

If construct
A control construct that executes a block of code if a logical expression evaluates to tRUE.

input
Information entered into the computer, usually from a keyboard or from a stored file. Also the process of entering that information.

insertion point
A blinking vertical bar that indicates where typed or pasted text will be inserted.

integer
A simple numeric data type consisting of whole numbers in the range: --32768 to 32767. The size of an integer is 2 bytes (16 bits). The declarator for this type is the percent sign (%). An integer has 4 significant digits.

integer expression
An numeric expression that contains an integer.

interpreted language
A language that is executed statement by statement, as opposed to a compiled program, in which all statements are translated before any are executed. Most BASICs are converted to tokens by the compiler and then interpreted.

landscape
Page orientation that aligns the paper's shorter dimension vertically with output flowing from top to bottom.

list box
A dialog box component that contains a list of available choices.

literal
A value expressed as itself rather than as the value of a variable or an expression.

local variable
A variable whose scope is limited to a given block of code, usually a subroutine.

logical expression
A mathematically constructed expression using reserved words such as AND and OR to define logical conditions.

logical operator
An operator that manipulates binary values at the bit level or manipulates true and false values. The logical operators are AND, OR, XOR, and NOT.

long
A simple numeric data type consisting of numbers in the long integer range: -2147483648 to 2147483647. The size of a long integer is 4 bytes (32 bits). The declarator for this type is the ampersand (&). A long has 9 significant digits.

loop
In programming, a process that repeats itself, usually until a variable condition returns a specified value.

macro
In the ScriptMaker Editor, a set of keystrokes and mouse movements that save time because they can be replayed by choosing Play Back Macro from the Edit menu. In the ScriptMaker language, a set of keystrokes and mouse movements that are recorded and then inserted into a script. They are replayed when the script is executed.

mask
A binary value used to selectively screen out or let through certain bits in a data value. Masking is performed by using a logical operator (AND, OR, XOR, NOT) to combine the mask and the data value.

mathematical expression
An expression that uses numeric values and operators, such as integers, fixed-point numbers, and floating-point numbers.

maximize
To enlarge a window to full-screen by using the maximize button (to the right of the title bar) or the maximize command from the Control menu.

Maximize button

A component of a window that zooms the window to full-screen size.

memory
Computer hardware that stores data and provides for retrieval of the data. Generally, the term memory refers to RAM, which is used to run applications as well as temporarily store data during program execution.

menu
In an application, a list of options that can be selected by the user. Menus typically include pull down, pop up, lists, or buttons. Choosing from a menu often leads to another menu or a dialog box containing further options.

menu bar
A vertical or horizontal section of the screen display that contains menu options in the form of words or icons that can be selected by the user.

menu-driven
Features and functions that are selected through some kind of menu, instead of or in addition to keystroke combinations.

menu item
A choice on a menu, selectable using either the keyboard or a mouse.

message
A piece of information passed from the application or operating system to the user to indicate a condition or suggest an action.

Microsoft Windows
An advanced software program that acts as a graphical user interface between the command line of a DOS-based machine and the user. *See* Windows.

minimize
To reduce a window to an icon on the desktop. You usually minimize windows when you want a process to run in the background while you do something else. Minimize a window by clicking the minimize button (also called *iconize*).

Minimize button

A component of a window that shrinks the window into a small icon.

mnemonic
A word, rhyme, or other memory aid used to associate a complex or lengthy set of information with something that is simple and easy to remember. Mnemonics are widely used in computing.

monospaced
Letters spaced so that every letter is given the same width of space (for example, the space for an "i" is as wide as the space for a "w").

modal dialog box
Dialog box that stops the script from executing statements until the user clicks one of the dialog boxs command buttons. All of ScriptMaker's predefined dialog boxes, except for the progress message dialog box, are modal.

modeless dialog box
Dialog box that allows the script to continue to execute statements while the dialog box is displayed. Only the progress message dialog box is modeless.

modulo
An arithmetic operation used in programming to find the remainder of a division operation.

mouse
A hand-held serial or bus device that translates motion from the users hand across a pad to a cursor on the computer screen.

mutual exclusion
A programming technique that ensures that only one program or routine at a time can access some resource. Also, allowed only one choice. For example, you can select only one option button from a group of option buttons.

nesting
A programming term that refers to embedding one construct inside of another.

NOT
An operator that performs Boolean (or logical) negation. In Boolean terms, NOT TRUE = FALSE and NOT FALSE = TRUE. In logical terms, if *value* contains a binary value, then NOT *value* changes each 0 bit to 1 and each 1 bit to 0.

null string
A string containing no characters and no location in memory. A string variable that has been assigned a null string requires no memory. This is not the same as an empty string.

numeric expression
An expression that evaluates to a value of type integer, long, single, or double. A numeric expression can be a numeric variable, numeric literal, numeric function, or any combination of these bound into one expression by numeric operators.

octal
The base-8 representation of a number. Used in programming as a way of representing binary numbers. More often used in microcomputers and mainframes than in personal computers.

operand
The object of a mathematical operation or a computer instruction. An operand can be data, or it can be the location in memory or on disk at which data is stored.

operator
In programming and computer applications, a symbol or other character indicating an operation that acts on one or more elements. Includes mathematical operators such as (+) and (-), logical operators such as AND and OR, relational operators such as (<) and (=), and so on.

operator precedence
The order in which the various operators in an expression are evaluated. Parentheses are often used to establish precedence. For example, (2 + 3) * 4 = 20; but 2 + (3 * 4) = 14.

option button (radio button)

A dialog box component that represents a mutually exclusive choice. Option buttons always appear with at least one other button, one of which is the default choice. Sometimes called a *radio button* by programmers.

OR
A logical operation combining two bits or two logical values. If one or both values are TRUE, it returns the value TRUE.

output
The results of computer processing. Output can be sent to the screen, a printer, a file, or to another computer in a network.

parameter
A variable or value that is passed from one routine to another.

parameter passing
In programming, the substitution of an actual parameter value for a formal, or dummy, parameter when a procedure or function call is processed.

parse
To break down an instruction into its component parts so the computer can act on it.

pass by reference
A means of passing a parameter to a subroutine. When a parameter is passed by reference, both the calling routine and the called routine use the same location in memory for the variable. Changes made to the value of the variable in the called routine change the value of the variable in the calling routine.

pass by value
A means of passing a parameter to a subroutine. When a parameter is passed by value, a copy of it is made for the called routine. Any changes made by the called routine affect only the value of the copy. They have no effect on the value of the variable in the calling routine.

password box
A text box that enters an asterisk for every character the user types, thus making it harder for an on-looker to read someones password.

path
A list of directories where DOS automatically searches for files when it cannot find requested files in the current directory. A PATH= statement is typically placed at the beginning of an AUTOEXEC.BAT file. One advantage of specifying a path is that programs located in directories listed in the path can be executed from any directory.

pathname
In a hierarchical filing system, a listing of the drive and/or directories that lead to a directory, file, or set of files. A complete pathname, such as C:\PICTURES\BITMAPS*.BMP, starts with the drive letter. A relative pathname, such as ..\LEVEL1*.DGN, specifies the location of a directory, file, or set of files in relation to the current directory.

pixel
A picture element. The smallest building block used to create an image.

point
To position the mouse pointer over an object (a window or menu, for example).

portrait
A page orientation that aligns the paper's longer dimension vertically with output flowing from top to bottom.

power
In mathematics, the number of times a value is multiplied by itself.

precedence
The order in which the various operators in an expression are evaluated. *See* operator precedence.

predefined function
A ScriptMaker function that has already been defined for you. For example, Len returns the length of a string. You never have to declare predefined functions.

predefined subroutine
A number of ScriptMaker statements are really predefined subroutines. For example the FileList statement can be executed in either of the following forms:

```
FileList files, "c:\*.bat"
```

Or,

```
call FileList (files, "c:\*.bat")
```

You never have to declare predefined subroutines.

primary mouse button
The mouse button you use the most. In most cases it is the left mouse button. *See* also secondary mouse button.

procedure
A series of logical steps that are used to accomplish a work-related objective or task. In programming, a set of computer program statements required to perform a task. A procedure is often called by another procedure. In ScriptMaker, this is called a subroutine.

program
A sequence of instructions that a computer can execute, including all the statements and files it needs.

program file
An executable file that launches an application. A program file has an .EXE, .PIF, .COM or .BAT extension. For example, NDW.EXE is the program file that launches Norton Desktop for Windows.

prompt button

The small box to the right of a drop-down list or combination box. When clicked, a list of choices appears.

programming language
An artificial language used to construct computer programs. Examples are BASIC, COBOL, C, Pascal, and others.

radian
In trigonometry, the length of the arc where a circle is intercepted by the two sides of an angle that begins in the center. Specifically, the unit of measure in which the length of the arc is equal to the radius of the circle.

random number generation
The creation of a number (or sequence of numbers) in a random or unpredictable order.

range
1) The spread between specified low and high values or beginning and ending times or dates.
2) In a spreadsheet, a block of contiguous cells selected for similar treatment.

read
For a computer, to collect information from an input source such as a file.

record
The basic structure of a database, consisting of a number of fields, each with its own name and type, and the contents of those fields. The elements of a record are accessed by their names.

Recorder
Accessed from the ScriptMaker Editor Tools menu, the macro recorder is used to record a series of events generated by the user within the Windows environment, then translate the recorded series of events into ScriptMaker statements. Those statements can be included in a script or subroutine that can reproduce the recorded series of events.

recursion
The ability of a routine or subprogram to call itself. Excessive recursion can eventually halt a program or even cause a system crash.

regular expression
When searching, a way to specify a range of possible matches. This is also known as using wildcards. For example, when searching for a DOS file, a question mark (?) represents any character, but only one character and an asterisk (*) represents any series of characters.
The ScriptMaker Editor has its own set of regular expressions:

[*chars*]	Any one of the characters between the brackets.
[~*chars*]	Any one of characters except for those specified between the brackets.
[char1 - char2]	Any one of the characters ranging from the first to the last specified character (in ASCII order).
@	Zero or more of the previous character.
% or <	The beginning of a line.
$ or >	The end of a line.
\t	A tab character (0x08 in hexadecimal).
\f	A formfeed character (0x0c in hexadecimal)
\char	The character. For example, \\ means \.

relational expression
An expression that uses a relational operator such as less than (<) or greater than (>) to express the relation between two or more values.

relational operator
An operator that allows the programmer to compare two or more values or expressions. Typical relational operators are <, >, =, <>, =>, <=.

replace
To put new data in the place of other data, usually after conducting a search for the data to be replaced. *See* search and replace.

reserved word
A word saved by DOS or an application for the programs own use. Reserved words cannot be used for naming variables, functions, and so on.

restore
Return a window to its previous size (its size before it was maximized or minimized).

return
To transfer control of the script from a called routine back to the routine that called it. Also, the value sent back to the calling routine from a function.

routine
A subroutine, function, or set of statements, such as those in an error-handling routine, that are separated from the rest of the subroutine or function in which they appear.

scope
The set of rules governing when and how an identifier can be accessed. These rules determine what variables a given routine can recognize and what other routines it can call.

script
A program written in the ScriptMaker language.

ScriptMaker
Is a BASIC programming language, that has tools for editing and testing programs, creating dialog boxes, and recording macros.

scroll arrow
One of the arrows found at either end of a scroll bar. *See* scroll bar.

scroll bar
The slider bar that appears along the right side or bottom of a window (or both) when the window contains more than it can display at one time. Clicking the scroll arrows or anywhere in the scroll bar moves the viewport up or down through the document.

scroll box
A small box that slides up and down in the scroll bar, indicating the relative position in a document or listing. Sometimes called a slider, elevator, or thumb.

scrolling
The process of moving a document in a window so you can see any part of it. So called because it is rather like reading a scroll.

search
To look for the location of a file, or to search a file or data structure for specific data. A search is carried out by a comparison or calculation to determine whether a match to some specified pattern exists.

search and replace
A word processing function, in which the user can specify two strings of charactersone string for the program to find and the other to replace the first string with.

search string
The string of characters to be matched in a search, typically a text string.

secondary mouse button
The mouse button you use the least. This is generally your right mouse button.

seed
The starting value used to generate random numbers.

select
In general computer use, to specify an item displayed on screen by highlighting or otherwise marking it, with the intent of manipulating the item in some way. Selecting generally indicates only that a choice has been made; a program does not act on a selection until instructed to do so.

separator line
A horizontal line that divides a menu into logical groups of menu items.

serial format
The date and time represented together as a double-precision, floating-point number whose value is the number of days since the zero date: December 30, 1899. The number of days precedes the decimal point and the time follows the decimal point. The time is represented as a fraction of a day. For example, July 4, 1993 is 34,154 days after December 30, 1899 and 12 noon is .5 days, so July 4, 1993 at 12 noon is represented by the number 34154.5. The serial format is useful for calculations on dates and times.

shell
An interface between you and DOS that makes managing your files easier. Norton Desktop for Windows is a graphical interface that acts as a shell.

show
Returns a hidden application to view.

sign
The character used to indicate a positive (+) or a negative (-) number.

significant digits
The sequence from the leftmost nonzero digit to the last nonzero digit in a decimal number, to the limit of the precision available.

simple data type
A data type that has only one part. For example, a number or a string is a simple data type because, while composed of several digits, bytes, or characters, each is operated on as an entity.

single
A simple numeric data type consisting of numbers in the range +/-3.4E+/-38. The size of a single-precision number is 4 bytes (32 bits: 1 for sign, 8 for the exponent, and 23 for the mantissa). The declarator for this type is the exclamation point (!). A double-precision number has 7 significant digits.

spin button
A dialog box component that consists of a text box coupled with two arrows. Use a spin button to cycle through a predetermined set of choices, often a set of numbers.

statement
The smallest executable part of a script. In general, each line of a script is a statement.

status bar
A component that displays information about a process, function or selected item. The status bar normally appears at the bottom of a window or dialog box.

string
Any sequence of consecutive characters, usually text.

string expression
An expression that evaluates to a value of type string. A string expression can be a string variable, string literal, string function, or any combination of these concatenated together.

string variable
An arbitrary name assigned by the programmer to a string. The programmer can then use or modify the string by referencing the string variable's name.

subprogram
A term use in some languages for *routine* (procedure or function) because the structure and syntax of a subprogram closely model those of a program.

subroutine
A group of statements that must be declared as a unit and perform some task. A subroutine is executed when its name is used in (or called from) another subroutine or function, but, unlike a function, its name does not return a value. Parameters are passed between the subroutine and the routine that called it.

subscript
In programming, a subscript is a number or variable that identifies an element in an array by its location. Each element of an array has a subscript for each dimension in the array. For example, Letters(2, 4) is the fourth element in the second row of the two-dimensional array named Letters. Subscripts are enclosed in parentheses and separated by commas.

substring
A section of a string. *See* string.

syntax error
Syntax errors are detected during compilation and occur when you make a mistake entering a command, such as not enclosing a string in quotes, or specifying the wrong number of parameters.

text box
A rectangular box (usually a single-line high) within a dialog box, into which you type the information needed to complete an action. It may be blank when it first appears, or it may contain text.

text file
A file composed of text characters, without formatting controls. Often refers to a word processing document. Useful for transferring files between word processing systems that could not otherwise read each other's documents. *See* ASCII file.

tile
To set open windows next to each other so that all windows are clearly visible.

title bar
The part of a window or dialog box that shows either the name (or title) of the application running in the window, or the label for the dialog box. The title bar in the currently selected, or *active*, window is a different color or intensity than the title bar in an inactive window.

toolbar
A row of command buttons displayed across the top of a window that provide quick mouse access to tools specific to the window containing the toolbar.

truncate
To cut off the beginning or end of a series of characters or numbers; in particular, to eliminate one or more of the least significant (typically rightmost) digits. In some editors, text that does not fit on a line is truncated when the file is saved. Database and spreadsheet programs often do this to fit a number into a smaller display space or cell.

tutorial
A teaching aid designed to help you learn to use a computer application. A tutorial may be a book or an interactive, disk-based series of self-paced lessons provided with the application.

twip
A twip is 1/20 of a point, so there are 1440 twips per inch.

two-dimensional
Existing in reference to two measurements, such as height and width. For example, a 2-dimensional array places numbers in rows and columns.

two-dimensional array
An array in which the location of any item is determined by two integers identifying its position in a particular row and column of a matrix.

type
In programming, *type* specifies the nature of a variable, such as integer, real number, or text character. Type can also refer to files and other data. *See* data type.

type declarator
A character placed after a variables name the first time it appears in a script to declare the variable to be of the particular type. The type declarator for an integer is %, for a long is &, for a single is !, for a double is #, and for a string is $.

user-defined constant
A constant that you create, declare, and use within a script. A user-defined constant can contain a string or numeric literal, one of the predefined constants TRUE and FALSE, or a previously declared user-defined constant.

user-defined dialog box
A dialog that you create, declare, and execute within a script. The statements that appear in the dialog box's declaration are executed when an instance of the dialog box is displayed.

user-defined error
An error that you create, declare, and trap within a script. You can also write error-handling routines that process the error when it occurs.

user-defined function
A function that you create, declare, and execute within a script. The statements that appear in the function's declaration are executed when the function is called.

user-defined subroutine
A subroutine that you create, declare, and execute within a script. The statements that appear in the subroutine's declaration are executed when the subroutine is called.

value
In programming and applications, a quantity assigned to a variable, symbol, label, or other element.

variable
The name of a location in memory that stores a value. The value of a variable can change during script execution. Every variable has a name, a data type, and a value.

variable expression
Any expression containing at least one variable. An expression that must be evaluated during program execution.

version
A number assigned by a software developer to identify the stage of program development. Successive public releases of a program have increasingly higher version numbers. Small changes are usually indicated by smaller increments, such as version 3.2 or 4.01. More significant changes usually take a full integer jump, such as from version 3.2 to version 4.0

viewport
The actual view into a document or image, which may include clipping or cutting off outer portions of an image that is larger than the viewport. When you resize a window while working on something else, the viewport is less than the full normal view in that window.

Viewport
A special window (in the Windows environment) opened from a script to display data.

wallpaper
A graphical image that is displayed on the desktop background.

wildcard
A global filename character that represents all or part of a filename. The question mark (?) represents any single character and an asterisk (*) represents a series of up to eight question marks.

WIN.INI
A file containing information and parameters to control your Windows environment. For example, WIN.INI stores the desktop color preferences you've chosen or the Windows applications you want to start automatically.

window
A framed area in which you can run an application, view a file listing or a document, or perform a task. A window can be opened, closed, resized and moved.

window corner
One of the four corners of a window used to resize the window.

window handle
Associated with each window is a handle, which is a number assigned to each window by Windows. A handle provides a more compact and efficient way to refer to a window.

Windows
The Microsoft Windows graphical environment. The Norton Desktop for Windows documentation set assumes Windows to be Microsoft Windows 3.0 or later.

write
To transfer information to an output device such as a printer or monitor, or to a file on disk. The opposite of read.

x-axis
The horizontal reference line on a chart or graph.

y-axis
The vertical reference line on a chart or graph.