Desktop Control Language

MCAFEE

Document Release DCL.10

Table of Contents

Chapter 1 Introduction **15**

An Overview of DCL....15
System Requirements....16
Viewing the Release Notes....16

Chapter 2 Getting Started **17**

A Sample DCL Script....17
Comments....18
Variables....19
Constants....21
Operators....22
Subroutines....23
Functions....23
Arrays....25
Conditional Statements....27
Loops....29

Chapter 3 Using the DCL Editor **31**

Starting the DCL Editor....31
Exiting DCL Editor....32
Creating a New Script....32
Using the Toolbar and Status Bar....32
Working With Files....33
Making an Executable....35
Printing a Script....37
Editing Text....38
Running a Script....41
Using DCL Tools....41
Recording a Macro....42
Working with Document Windows....48
Getting Help....48
Text Editing Keys....49

Chapter 4 Using the Dialog Editor 51

Starting the Dialog Editor 51
Creating or Modifying a Dialog Box Template 53
Using the Toolbar 53
Using the Status Bar 54
Adding Dialog Box Controls 54
Defining Dialog Box Controls 55
Cutting, Copying and Pasting Controls 59
Capturing Dialog Boxes 59
Opening an Existing File 60
Saving a New Dialog Box Template 61
Saving an Existing File 61
Running the Dialog Editor Standalone 62
Getting Help 62
Sample Script 63
Exiting the Dialog Editor 65
Shortcut Keys 65

Chapter 5 Using the Debugger 67

Starting the Debugger 67
Running a Script 68
Using the Toolbar 68
Tracing Through a Script 69
Setting Breakpoints 69
Using Watch Variables 69
Editing Text 71
Editing Dialogs 74
Getting Help 74
Exiting the Debugger 75
Shortcut Keys 75

Chapter 6 DCL Characteristics 77

General Characteristics 77
Script Execution 78
Structures 78
Compiler 78
Variables 78
Expression Evaluation 79
Subroutines & Functions 80
Error Handling 80
Arrays 81
Data Types 81

Operator Precedence 83
DCL Comments 83
Constants 84
DCL Limitations 85

Chapter 7 Related Commands 87

Arrays 87
Clipboard Manipulation 87
Conversions 88
Date and Time Functions 88
DCL Environment Information 89
Desktop Modifications 89
Dialog Creation 89
Dialog Display 90
Dialog Manipulation 90
Dynamic Data Exchange (DDE) 91
Environment Statements and Functions 92
Error Trapping 92
File Input and Output 93
Flow Control 94
Icons 94
Keyboard Manipulation 95
Math Statements and Functions 95
Menus 96
Miscellaneous Statements and Functions 96
Mouse Events 96
Network Functions 97
Operators 97
Printer Manipulation 98
Procedure Statements 98
Strings 98
Variables and Constants 100
Viewport Window Manipulation 100
Window Manipulation 100

Chapter 8 Command Reference 103

' 104
* 104
+ 104
- 104
/ 105
< 105
<= 105
<> 106
= 106
> 106
>= 107
\ 107
^ 107
Abs Function 107
ActivateControl Statement 108
AddIni Function 109
And 109
AnswerBox Function 110
AppActivate Statement 111
AppClose Statement 112
AppFileName$ Function 112
AppFind Function 113
AppGetActive$ Function 113
AppGetPosition Statement 114
AppGetState Function 115
AppHide Statement 116
AppList Statement 117
AppMaximize Statement 117
AppMinimize Statement 118
AppMove Statement 119
AppRestore Statement 120
AppSetState Statement 121
AppShow Statement 121
AppSize Statement 122
AppType Function 123
ArrayDims Function 124
ArraySort Statement 125
Asc Function 125
AskBox$ Function 126
AskPassword$ Function 127
Atn Function 127
ATTR_ARCHIVE 128
ATTR_DIRECTORY 128

ATTR_HIDDEN 128
ATTR_NONE 128
ATTR_NORMAL 129
ATTR_READONLY 130
ATTR_SYSTEM 130
ATTR_VOLUME 130
Beep Statement 130
Begin Dialog...End Dialog Statement 131
ButtonEnabled Function 132
ButtonExists Function 132
Call Statement 133
CancelButton Statement 134
CDbl Function 134
ChDir Statement 135
ChDrive Statement 135
CheckBox Statement 136
CheckboxEnabled Function 137
CheckboxExists Function 137
Chr$ Function 137
CInt Function 138
Clipboard$ Statement and Function 139
ClipboardClear Statement 139
CLng Function 140
Close Statement 140
Combobox Statement 141
ComboboxEnabled Function 141
ComboboxExists Function 142
Command$ Statement 142
Const Statement 143
Cos Function 143
CSng Function 143
CStr Function 144
CurDir$ Function 144
Date$ Statement and Function 145
DateSerial Function 146
DateValue Function 146
Day Function 147
DCLHomeDir$ Function 147
DCLOS$ Function 148
DCLVersion$ Function 148
DDEExecute Statement 149
DDEInitiate Function 149
DDEPoke Statement 150
DDERequest Function 150
DDETerminate Statement 151

DDETerminateAll Statement 151
DDETimeOut Statement 151
DDE Example 152
Declare Statement 154
DEFtype Statement 156
DesktopCascade Statement 157
DesktopSetColors Statement 157
DesktopSetWallpaper Statement 158
DesktopTile Statement 159
Dialog Examples 160
Dialog Statement and Function 163
Dim Statement 164
Dir$ Function 167
DirExists Function 167
DiskDrives Statement 168
DiskFree Function 168
Do...Loop Statement 169
DoEvents Statement 171
DoKeys Statement 172
EditEnabled Function 172
EditExists Function 173
EnableStopScript Statement 173
End Statement 174
ENV_BOTH 174
ENV_DOS 175
ENV_WINDOWS 175
Environ$ Function 175
EOF Function 176
Erl Function 176
Err Statement and Function 176
Error Statement 177
Error$ Function 178
Exclusive Statement 179
Exit Do Statement 180
Exit For Statement 180
Exit Function Statement 181
Exit Sub Statement 181
Exit Statement Examples 181
Exp Function 183
FALSE 183
FileAttr Function 183
FileCopy Function 184
FileDateTime Function 185
FileDirs Statement 186
FileExists Function 186

FileLen Function....187
FileList Statement....188
FileParse Statement....189
FileType Function....190
FindFile$ Function....191
Fix Function....192
For...Next Statement....193
FreeFile Function....195
Function...End Function Statement....195
GetAttr Function....196
GetCheckbox Function....197
GetComboboxItem$ Function....198
GetComboboxItemCount Function....198
GetEditText$ Function....199
GetEnv Function....199
GetListboxItem$ Function....200
GetListboxItemCount Function....201
GetOption Function....201
GetUserName Function....202
GoSub Statement....202
Goto Statement....203
GroupBox Statement....203
Hex$ Function....204
HLine Statement....204
Hour Function....205
HPage Statement....206
HScroll Statement....206
If...Then...Else Statement....207
Input # Statement....208
Input$ Function....209
Input/Output Example....209
InputBox$ Function....211
InStr Function....212
Int Function....214
Item$ Function....214
ItemCount Function....215
Kill Statement....215
LBound Function....216
LCase$ Function....216
Left$ Function....217
Len Function....218
Let Statement....218
Line$ Function....219
LineCount Function....220
LineInput # Statement....220

ListBox Statement 221
ListboxEnabled Function 221
ListboxExists Function 222
Loc Function 222
LOF Function 222
Log Function 223
LTrim$ Function 223
Main Statement 224
MCI Function 224
Menu Statement 225
MenuItemChecked Function 226
MenuItemEnabled Function 226
MenuItemExists Function 226
Mid$ Function 227
Minute Function 228
MkDir Statement 229
Mod 229
Month Function 230
Message Example 231
MsgBox Statement and Function 231
MsgClose Statement 235
MsgOpen Statement 235
MsgSetText Statement 237
MsgSetThermometer Statement 237
Name Statement 237
NetAttach Function 238
NetConnectDrive Function 239
NetDetach Function 239
NetDirectoryRights Function 240
NetDisconnectDrive Function 241
NetGetDirectoryRights$ Function 242
NetMemberOf Function 242
NetStationID Function 244
NetUserName Function 244
NetworkStatus Function 245
Not 246
Now Function 247
NS_ACTIVE 248
NS_LOGGEDON 248
Null Function 248
Oct$ Function 249
OKButton Statement 249
On Error Statement 249
Open Statement 250
OpenFileName$ Function 251

Option Base Statement....252
OptionButton Statement....252
OptionEnabled Function....253
OptionExists Function....253
OptionGroup Statement....254
Or....254
PI....255
PO_LANDSCAPE....256
PO_PORTRAIT....256
PopupMenu Function....257
Print Statement....258
Print # Statement....258
PrinterGetOrientation Function....259
PrinterSetOrientation Statement....260
PrintFile Function....261
PushButton Statement....262
QueEmpty Statement....262
QueFlush Statement....263
QueKeyDn Statement....263
QueKeys Statement....264
QueKeyUp Statement....265
QueMouseClick Statement....266
QueMouseDblClk Statement....266
QueMouseDblDn Statement....267
QueMouseDn Statement....267
QueMouseMove Statement....268
QueMouseUp Statement....268
QueSetRelativeWindow Statement....268
Queue Example....269
Random Function....271
Randomize Function....271
ReadINI$ Function....272
ReadINISection Statement....272
ReDim Statement....273
REM Statement....274
RefreshIni Statement....275
Reset Statement....275
RestoreEnv Function....275
Resume Statement....276
Return Statement....277
Right$ Function....278
RmDir Statement....278
Rnd Function....279
RTrim$ Function....279
SaveEnv Function....280

SaveFileName$ Function 280
Second Function 281
Seek Statement and Function 281
Select...Case Statement 282
SelectBox Function 283
SelectButton Statement 284
SelectComboboxItem Statement 284
SelectListboxItem Statement 285
SendKeys Statement 286
SetAttr Statement 286
SetCheckbox Statement 287
SetEditText Statement 288
SetEnv Function 289
SetIcon Statement 290
SetIconTitle Statement 290
SetOption Statement 290
Sgn Function 291
Shell Function 291
ShowIcon Statement 293
Sin Function 293
Sleep Statement 294
SleepUntil Function 294
Snapshot Statement 296
Space$ Function 297
Sqr Function 297
Stop Statement 298
Str$ Function 298
StrComp Function 299
String$ Function 300
StringSort Function 301
Sub...End Sub Statement 301
SystemFreeMemory Function 302
SystemFreeResources Function 302
SystemMouseTrails Statement 302
SystemRestart Statement 303
SystemTotalMemory Function 303
SystemWindowsDirectory$ Function 304
SystemWindowsVersion$ Function 304
Tan Function 305
Text Statement 305
TextBox Statement 305
Time$ Statement and Function 306
Timer Function 307
TimeSerial Function 307
TimeValue Function 308

Trim$ Function 308
TRUE 309
TYPE_DOS 309
TYPE_WINDOWS 309
UBound Function 309
UCase$ Function 310
Val Function 310
ViewportClear Statement 311
ViewportClose Statement 312
ViewportOpen Statement 313
VK_LBUTTON 314
VK_RBUTTON 315
VLine Statement 315
VPage Statement 316
VScroll Statement 316
WaitForTaskCompletion Function 317
Weekday Function 318
While...Wend Statement 319
WinActivate Statement 319
WinClose Statement 320
WinFind Function 321
WinList Function 321
WinMaximize Statement 322
WinMinimize Statement 323
WinMove Statement 323
WinRestore Statement 325
WinSize Statement 326
Word$ Function 326
WordCount Function 327
Write # Statement 328
WriteINI Statement 328
WS_MAXIMIZED 329
WS_MINIMIZED 329
WS_RESTORED 329
Xor 329
Year Function 330

Index **331**

Notes

Chapter 1 Introduction

Welcome to Desktop Control Language™ (DCL), a full-featured, yet easy-to-use scripting language which gives you greater control over your Microsoft® Windows™ network environment. This chapter contains an overview of DCL, system requirements for running DCL and information about your package.

An Overview of DCL

DCL is built on the easy-to-learn Basic language. It provides all of the features of a major programming language, including subroutines, functions, strings, arrays, numeric support and advanced flow control.

DCL provides an integrated development environment for editing, testing and debugging scripts using the following features:

Feature	Description
Macro Recorder	Allows you to automatically generate scripts to control other Windows applications. Furthermore the code can be modified as needed.
Syntax Checker	Reads the scripts for entry errors.
Debugger	Allows you to set breakpoints, watch variables and trace through scripts for logic errors.
Dialog Editor	Allows you to design dialog boxes and define their controls as well as capture dialog boxes from other applications and paste them into the Dialog Editor.

When you have finished writing and testing your script, you can build an executable (.EXE) file for distribution.

Note: DCL translates any MultiSet scripts you run into DCL scripts. The conversion is automatic.

System Requirements

The following list contains the minimum system requirements to run this version of DCL.

- An 80386/SX or higher based computer.
- 4 megabytes of memory.
- One 1.44 MB (3-1/2") floppy disk drive (for installation).
- MS-DOS Version 5.0 or later.
- Microsoft Windows Version 3.1 or later, or Windows for Workgroups Version 3.1 or later.
- A monitor and VGA graphics card or other high-resolution graphics card compatible with Windows Version 3.1 or later.
- A Microsoft Windows-compatible mouse, recommended but not required.

Viewing the Release Notes

Note: The Release Notes contain important information on the successful operation of the product as well as its latest enhancements. Please read the release notes before continuing with the installation and implication of this product.

The release notes for DCL are located on Disk 1. The release note file exists in Windows Write format.

Use the following procedure to view the release notes.

1. Choose File | Run.

   The Run dialog box is displayed.

2. Enter WRITE.EXE in the text box.

   An untitled .WRI file opens.

3. Choose File | Open.
4. Select READDCL.WRI and choose OK.

This file is copied to the default directory during installation.

Chapter 2 Getting Started

This chapter will introduce you to the elements that make up DCL's implementation of Basic. If you have had Basic programming experience, we recommend that you peruse this chapter as well as Chapter 6, "Desktop Control Language Characteristics," as different implementations of Basic may vary. If you are new to Basic programming, this chapter will introduce you to the fundamentals of the language.

For further information refer to Chapters 7 and 8, "Related Commands" and "Command Reference, " respectively.

A Sample DCL Script

Below is a sample DCL script:

```
sub main()
    msgbox "Hello World"
end sub
```

Each DCL script contains a subroutine called `main`. A subroutine is a set of commands that perform a specific task. Simple Basic programs often have only one subroutine. More complex Basic programs often have more than one subroutine.

The commands `sub main` and `end sub` define the beginning and end of the subroutine. The above sample subroutine contains only one statement `msgbox "Hello World"`.

Running the Script

Use the following procedure to run the sample script.

1. Start the DCL Editor (DCLEDIT.EXE).
2. Enter the second line of the above script.

   The first and third lines are already entered for you.

3. Choose Run | Start Script.

   The sample message box is displayed.

Figure 2-1: Hello World message box

4. Choose OK to return to the DCL Editor window.

Saving the Script

Use the following procedure to save the script.

1. Choose File | Save.

   The File Save As dialog box is displayed.

2. Enter Hello in the File Name box.

3. Choose OK.

   A text (ASCII) file named HELLO.DCL contains your script. The default extension for all scripts is .DCL.

Comments

A comment is a note you write in a script. It is not executed with your script. Its sole purpose is to provide information about the script. Comments are very useful when you have to modify a script.

A comment begins with a single quote (') and continues for the rest of the line. The following script contains comments:

```
sub main()' Script Name: Hello
' Written: 3/1/94
' Author: J. Doe
    msgbox "Hello World"    'display message box
end sub
```

A comment can occupy the same line as a DCL command if it follows the command.

Variables

In the previous script, we provided literal text (the actual letters, digits, and special characters) to be displayed in the message box. Literal text is enclosed in double quotation marks (""), for example, `"Hello World"` . Although useful, literal text has its limitations. For example, what if we wanted to say "Hello Everybody" instead of "Hello World." We would have to write a new script.

Variables allow you to change the data displayed by and used in your scripts. A variable is a name with which you associate data—a string of characters or a number. Variable names can be up to 40 characters long. They must start with a letter and can contain letters and digits.

Data Types

The most commonly used types of data in scripts are strings and integers. DCL supports many data types, allowing it to accommodate almost any programming situation. These data types include integers, long integers, strings, dialogs, single precision floats and double precision floats. For further information regarding data types refer to Chapter 6, “Desktop Control Language Characteristics.”

Strings

A string is a succession of letters, digits, spaces and special characters as listed below:

"Hello"
"Hello "
"Hello World"
"Testing 1 2 3"
"" (Null string)

A string variable is simply a name representing a string. The actual string of characters represented by the variable may change. The following script makes use of a string variable:

```
sub main()
    name$ = InputBox$("Enter your name:")
    msgbox "Hello " + name$
end sub
```

`InputBox$` is a function which displays a dialog box which prompts users to enter their names. The names entered by the users are stored in a string variable called `name$`. The contents of the string variable `name$` might be "Jack" or "Jane" or "Chris". The dollar ($) symbol at the end of a variable name indicates that the variable is a string.

The `msgbox` statement displays a string that is made up of the literal "Hello " and the contents of the string variable `name$`. The two strings are combined (concatenated) by the '+' operator.

The following dialog boxes are generated by the above script.

Figure 2-2: Input box

Assume that the user typed 'Chris' and chose OK.

The following message is displayed:

Figure 2-3: Hello Chris message box

Integers

Integers are whole numbers between -32,768 and 32,767. Long Integers are whole numbers between -2,147,483,648 and 2,147,483,647. The following script uses integers:

```
sub main()
    SysRes%=SystemFreeResources
    FreeMem&=SystemFreeMemory
end sub
```

The `SystemFreeResources` function returns an integer between 0 and 100, indicating the percentage of free system resources. This value is stored in the integer variable `SysRes%`. An integer variable is indicated by the `%` symbol at the end of the name.

The `SystemFreeMemory` function returns a long integer representing the amount of free memory. The value is stored in the long integer variable `FreeMem&`. The `&` symbol at the end of a variable name indicates a long integer.

Variable names are not case-sensitive. However, you can use capitalization to make your variable names more readable. For example, `FreeMem&` is much more readable than `freemem&`.

Constants

A constant is a value that does not change such as 0, 1, 2, 4, etc. To help make your scripts more readable, DCL provides symbolic constants, text strings which represent numeric values. For example, the constant FALSE equals 0 and TRUE equals -1. You do not need to know the numeric value of a constant in order to use it.

The following script illustrates the use of symbolic constants:

```
sub main()
    System MouseTrails TRUE
    msgbox "Move the Mouse Around Now"
    SystemMouseTrails FALSE
end sub
```

The values TRUE and FALSE are used to toggle the display of mouse trails on and off.

Several DCL commands make use of symbolic constants. By convention, constant names are in uppercase. For further information refer to Chapter 8, “Command Reference.”

User-Defined Constants

The `Const` command allows you to define your own constants.

The following script converts from Fahrenheit to Centigrade. The formula for converting from Fahrenheit to Centigrade is:

Centigrade=((Fahrenheit - 32) * 5) / 9

The * is the symbol for multiplication. The / represents division.

The script performs the conversion, using the constant `FREEZING` instead of the literal 32.

Note: Constants must be defined before the `main` subroutine is executed.

In the following script, the `val` function converts a string to a number, the `str$` function converts a number to a string.

```
Const FREEZING = 32
sub main()
    ftemp$ = InputBox$("Enter a F  ahrenheit temperature.")
    fahrenheit% = val(ftemp$)
    centigrade% = ((fahrenheit% - FREEZING) * 5) / 9
    ctemp$ = str$(centigrade%)
    msgbox "Centigrade temperature:" + ctemp$
end sub
```

Operators

The following operators are used by DCL and listed in highest to lowest precedence:

Operator	Description
()	parentheses
^	exponentiation
-	unary minus
/, *	multiplication and division
\	integer division
mod	module
+, -	addition and subtraction
=, <>, >, <, <=, >=	relational
not	logical negation

and	and
or	or
xor	exclusive or

Subroutines

A subroutine is a separate block of DCL code (in the same script) that is called and executed as a unit. Short scripts usually do not require subroutines. When you write a long script, it is helpful to break tasks down into subroutines. The following code illustrates a subroutine:

```
sub HelloProcessing()
    name$ = InputBox$("Enter your name:")
    msgbox "Hello " + name$
end sub
sub main()
    HelloProcessing
    'Rest of Script
end sub
```

In this example the subroutine HelloProcessing greets users and prompts them for their names. The block of code contained in the subroutine begins with the command `sub <NameOfSubroutine>` and ends with the command `end sub`.

To call a subroutine, enter the name of the subroutine as you would any other command. The "command" `HelloProcessing` causes the HelloProcessing subroutine to be executed.

Note: The subroutine must be defined before the call to the subroutine.

For further information regarding subroutines, refer to Chapter 8, "Command Reference."

Functions

Functions are like subroutines. Unlike subroutines, however, functions return values (integers, strings, etc.). `InputBox$` is a DCL function that returns a string. When a DCL function is called, the returned value in an expression is used and any data to be manipulated by the function must be provided. The term for this data is arguments or parameters. The following is a sample of a function's general format:

VaraibleName = Function(Parameter1,Parameter2,etc.)

For example:

```
name$ = InputBox$("Enter your name:")
```

In this code sample, `name$` is a string variable that contains the string value returned by `InputBox$`. The argument "Enter your name:" is passed to the function and displayed in the dialog box generated by `InputBox$`.

The `len` function returns an integer representing the length of a string. For example:

```
StringLength%=len(name$)
```

In this example, the `len` function puts the length of the string `name$` in the integer variable `StringLength%`.

User-Defined Functions

You can write your own functions. Functions are useful for frequent tasks that return a value. The following script uses a function to convert from Fahrenheit to Centigrade.

```
Const FREEZING = 32
function FarToCent%(f%)
    FarToCent% = ((f% - FREEZING) * 5) / 9
end function
sub main()
    ftemp$ = InputBox$("Enter a Fahrenheit temperature.")
    fahrenheit% = val(ftemp$)
    centigrade% = FarToCent%(fahrenheit%)
    ctemp$ = str$(centigrade%)
    msgbox "Centigrade temperature:" + ctemp$
end sub
```

The function block is defined by the function and end function statements.

The `function` statement contains the function name, return type and parameters passed to the function. The function name (`FarToCent%`) ends in a character that indicates the type of value the function returns. The function in the above script returns an integer (denoted by the % symbol). One parameter (`f%`) is passed to the function.

The value to be returned is assigned to the function name (i.e., `FarToCent%`).

The processing of the function occurs as follows:

1. In the `main` subroutine, `FarToCent%(fahrenheit%)` calls the `FarToCent%` function and passes it the `fahrenheit%` parameter.
2. The `FarToCent%` function receives the integer parameter as `f%` and uses it to represent the Fahrenheit temperature in the conversion to Centigrade.
3. The `FarToCent%` function performs the conversion and returns the result.
4. The returned value is put in the `Centigrade%` variable in the `main` subroutine.

Passing Parameters by Reference and by Value

As a default, parameters are passed to functions by *reference*. Meaning that the function-side parameters refer to the same variable as the calling-statement-side parameters. For example, `fahrenheit%` and `f%` refer to the same variable. If you were to change `f%` in the `FarToCent%` function `fahrenheit%` in the `main` subroutine would also be changed.

The alternative to passing parameters by reference is to pass them by *value*. Meaning that the *contents* of the variable are passed to the function. The function-side parameter and the calling-statement-side parameter refer to different variables. To specify that a parameter is to be passed by value, precede the parameter in the `function` statement with `byval`. By adding `byval` to the `FarToCent%` function, you can cause `f%` and `fahrenheit%` to be entirely separate variables. You can change `f%` in the function without affecting `fahrenheit%` in the `main` subroutine.

```
Const FREEZING = 32
function FarToCent%(byval f%)
    FarToCent% = ((f% - FREEZING) * 5) / 9
end function
sub main()
    ftemp$ = InputBox$("Enter a Fahrenheit temperature.")
    fahrenheit% = val(ftemp$)
    centigrade% = FarToCent%(fahrenheit%)
    ctemp$ = str$(centigrade%)
    msgbox "Centigrade temperature:" + ctemp$
end sub
```

For further information regarding functions, refer to Chapter 8, “Command Reference.”

Arrays

An array is a series of variables of the same type all having the same name. You differentiate between the elements of the array by using an integer index. For example, assume that you have an array of strings called Cities$.

Cities$	Index
"Atlanta"	0
"Chicago"	1
"Los Angeles"	2
"Montreal"	3
"New York"	4

Cities$ array

`Cities$(0)` contains the string "Atlanta"; `Cities$(1)` contains "Chicago", etc. The form for referencing an element of an array is:

ArrayVariableName(Index)

Below is a sample script that uses an array of strings:

```
sub main()
    dim Performance$( 5) as String
    Performance$(0)="Poor"
    Performance$(1)="Fair"
    Performance$(2)="Good"
    Performance$(3)="Excellent"
    Performance$(4)="Perfect"
    gradestr$=InputBox$("Enter a grade from 0 to 100.")
    grade#=val(gradestr$)
    plevel%=grade#\25
    msgbox "The grade is " + Performance$(plevel%)
end sub
```

The activities required to use an array are:

- Declare the array using the `dim` statement. This statement names the array, indicates the number of elements (and dimensions) and specifies the type of data to be contained in the array. The sample script defines an array named `Performance` of type `string`, containing 5 elements and one dimension.

- Load the array with values. This essentially amounts to assigning a value to each element in the array. The `Performance(n)="value"` statements load the array. The number in parentheses is the index to the array. (Some DCL functions that use arrays load the initial values for you.) By default the first element of an array has 0 as its index. You can use the `Option Base` to set this value to 1, if desired.
- Reference the array as needed. In the subsequent code, an input box asks the user to enter a number from 0 to 100. This returns a string, which the `val` function converts into the long integer `grade#`. We perform integer division of `grade#` by 25 to get an integer between 0 and 4. (Integer division returns only the nonfractional part of the quotient.) This integer is `plevel%`, which serves as an index to select the correct element from the array. The selected string is then displayed in a message box.

For further information regarding arrays, refer to Chapter 8, “Command Reference.”

Conditional Statements

Conditional statements are executed only if a particular condition is true. DCL supports the following conditional statements: `If...Then...Else` and `Select Case`.

If...Then...Else

The `if...then` statement performs a statement (or set of statements) if a condition is true. Otherwise the statement is skipped. Below is an example of the `if...then` statement.

```
sub main()
    SysRes%=SystemFreeResources
    if SysRes% < 30 then
        msgbox "Your available resources are low."
    end if
end sub
```

This script informs the user if available system resources fall below 30 percent. The `if` keyword specifies the condition that must be true; the `then` keyword signals the beginning of one or more statements to be executed if the condition is true. The terms `if` and `then` must be on the same line. The `if...then` statement ends with the `end if` keyword.

The `if...then...else` statement allows you to specify activities for the script to perform whether the condition is true or false. The `else` keyword precedes one or more statements to be executed when the condition is false.

```
sub main()
    SysRes%=SystemFreeResources
    if SysRes% < 30 then
        msgbox "Your available resources are low."
    else
        msgbox "Your available resources are fine."
    end if
end sub
```

A more complex form of the `if...then...else` statement allows you to accommodate multiple conditions. The following `if...then...elseif...else` statement varies the message depending on the percentage of available system resources.

```
sub main()
    SysRes%=SystemFreeResources
    if SysRes% < 30 then
        msgbox "Your available resources are low."
    elseif SysRes% < 20 then
        msgbox "Resources extremely low. Close some Applications now."
    elseif SysRes% < 10 then
        msgbox "Resources are DANGEROUSLY low. Close Applications now."
    else        'everything's ok
        msgbox "Your available resources are fine."
    end if
end sub
```

In this manual the symbol ↵ indicates we had to float the rest of a DCL command to the next line. In DCL scripts, a command cannot be broken between lines. In some cases, the book size made it impossible to keep all of the code on the same line.

For further information, refer to Chapter 8, “Command Reference.”

Select Case

The `select case` statement also handles multiple conditions.

Like the previous `if...then...elseif...else` statement, the following `select case` statement varies the message depending on the percentage of available system resources.

```
sub main()
    SysRes%=SystemFreeResources
    select case SysRes%
        case 21 to 30
            msgbox "Your available resources are low."
        case 11 to 20
            msgbox "Resources extremely low. Close some Applications now."
        case 0 to 10
            msgbox "Resources are DANGEROUSLY low.   ↵
                Close Applications now."
        case else       'everything's ok
            msgbox "Your available resources are fine."
    end select
end sub
```

The `select case` statement begins with `select case <VariableName>` and ends with `end select`. The logic of the `select case` statement is to test

each `case` in the `select case...end select` block. Each `case` statement specifies a condition. If the condition is true, the statement(s) associated with are executed. The `case else` statement is executed if none of the conditions tested are true. The breadth of expressions supported by the case statement make it a powerful tool for conditional processing.

For further information, refer to Chapter 8, “Command Reference.”

Loops

Loops are blocks of statements which perform the same action multiple times while a condition is true or until a condition is met. Each pass through a loop is called an iteration. DCL supports the following iterative statements: `For...Next`, `While...Wend`, and `Do...Loop`.

For further information, refer to Chapter 8, “Command Reference.”

For...Next

A `For...Next` loop executes a group of statements a specified number of times. Below is a sample `For...Next` loop:

```
sub main()
    for x% = 1 to 5 step 1
        msgbox str$(x%)
    next x%
    msgbox "You're past the loop."
end sub
```

In the `For` statement you specify a:

- Counter variable, which in our example is `x%`.
- Beginning and ending value for the counter variable, in our example 1 and 5 respectively.
- Increment. This is provided by the `step` portion of the statement. The `step` can be omitted if the increment is 1.

The `Next` statement marks the end of the loop and increments the counter variable by the `step` value.

The statements inside the loop are executed repeatedly until the counter exceeds the ending value. In our example the loop is executed five times. When the counter exceeds the ending value, the script continues with the next statement after the `For...Next` loop.

For example. use the variables to represent starting and ending values:

```
sub main()
    startval%=1
    endval%=10
    incr%=2
    for x% = startval% to endval% step incr%
        msgbox str$(x%)
    next x%
    msgbox "You're past the loop."
end sub
```

While...Wend

A `While...Wend` loop executes a group of statements repeatedly while a condition is true. Once the condition is false, the statement following the `Wend` statement is executed.

```
sub main()
    testval%=1
    while testval% < 6
        msgbox str$(testval%)
        testval% = testval% +1
    wend
    msgbox "You're past the loop."
end sub
```

The `While` statement marks the beginning of the loop and specifies the test condition. In the above example, the condition is that the variable `testval%` must be less than 6.

The `Wend` statement marks the end of the loop. The loop is executed repeatedly until `testval%` is equal to 6.

Do...Loop

The Do...Loop statement executes a group of statements while or until a condition is true. It has several forms.

Chapter 3 Using the DCL Editor

DCL provides a multiple document interface (MDI) to ease the process of editing, testing and debugging scripts. This chapter describes how to use the DCL Editor to create, modify and test your scripts.

Starting the DCL Editor

To start the DCL Editor (DCLEDIT.EXE), choose the DCL Editor icon from the Applications Manager window (or Program Manager window).

The DCL Editor window is displayed.

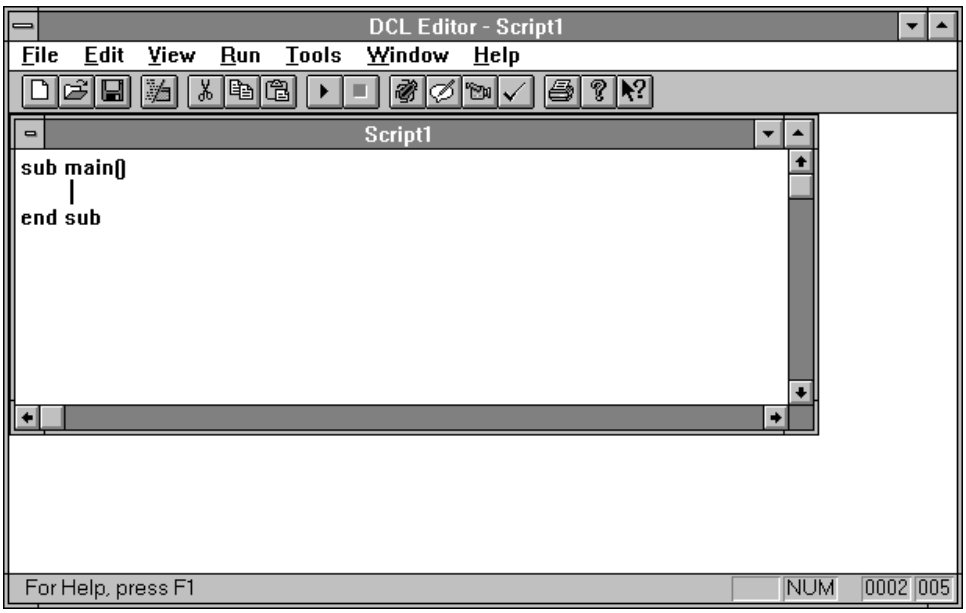


Figure 3-1: The DCL Editor window

Exiting DCL Editor

To exit DCL Editor, choose File | Exit.

Creating a New Script

Use the New command to open a window and the following procedure to create a new script.

1. Choose File | New.
2. Enter the commands that make up the script.

   If you need help for a particular command, position the cursor on the line containing the command and press F1.
3. Save your script as described under “Saving a New Script” in this chapter.
4. Run your script as described under “Running a Script” in this chapter.

Using the Toolbar and Status Bar

DCL provides a Toolbar and Status Bar for your convenience in developing scripts.

The Toolbar is a row of buttons representing frequently used DCL Editor commands. Instead of using the menus, you can choose a Toolbar button to execute a command.

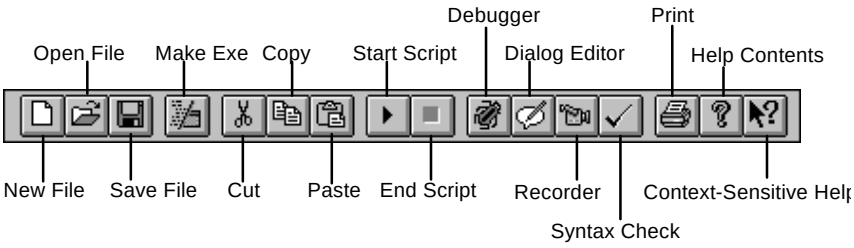


Figure 3-2: The DCL Editor Toolbar

To toggle the Toolbar display on or off, choose View | Toolbar.

Using the Status Bar

The Status Bar appears at the bottom of the DCL Editor window.

When a menu is displayed, the Status Bar provides a description of the currently selected item.

The Status Bar also indicates when the Caps Lock or Num Lock key is on. The current line and column of the cursor are displayed in far right corner of the Status Bar.

To toggle the Status Bar display on or off, choose View | Status Bar.

Figure 3-3: The DCL Status Bar

Working With Files

The scripts you write are saved as ASCII (text) files.

Opening an Existing File

Use the following procedure to open an existing file.

1. Choose File | Open.

   The File Open dialog box is displayed.

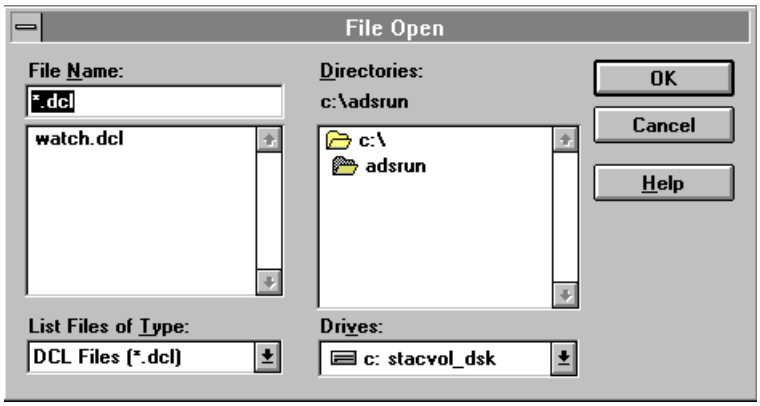


Figure 3-4: The File Open dialog box

2. Select the Drive and Directory containing the file.

3. Enter the File Name. Or select the file from the File Name list box.

   You can change the types of files displayed in the File Name list box by selecting another file type from the List Files of Type box.

4. Choose OK.

Opening a Recent File

The last four files you saved are displayed toward the bottom of the File menu.

To open one of these files, choose the file name from the File menu.

Closing a File

Use the Close command to close the current script. If you have made changes to the file, you will be asked whether you want to save the changes.

Saving a New Script

Use the following procedure to save the current new script.

1. Choose File | Save.

   The File Save As dialog box is displayed.

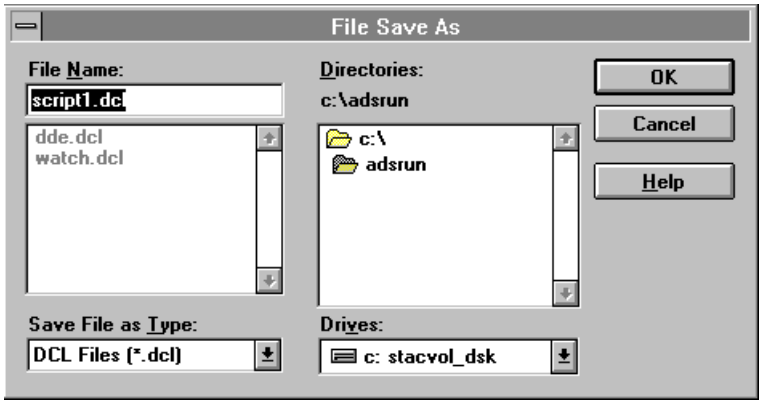


Figure 3-5: The File Save As dialog box

2. If you need to change the current path, select the Drive and Directory to contain the file.

3. Enter the File Name.

4. Choose OK.

Saving an Existing File

Use the following procedure to save an existing file.

1. Choose File | Save As.

   The File Save As dialog box is displayed.

2. If you want to change the current path, select the Drive and Directory to contain the file.

3. Enter the File Name.

4. From the Save File As Type box, select the desired file type.

5. Choose OK.

   If a file having the same name already exists, you will be asked to confirm that you want to overwrite the file.

Making an Executable

Use the following procedure to save the current script as an executable file.

1. Choose File | Make Exe.

   The Make Executable dialog box is displayed.

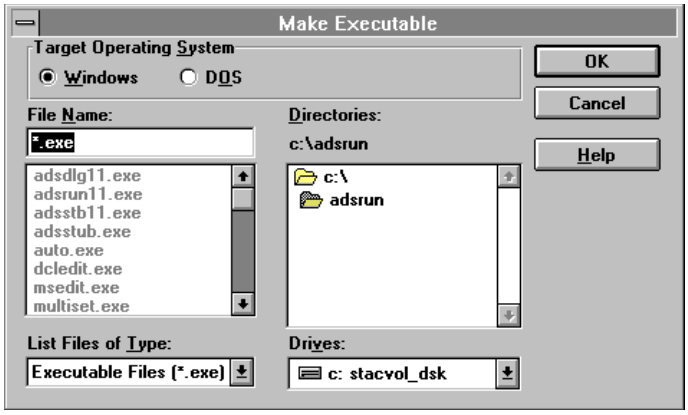


Figure 3-6: The Make Executable dialog box

2. Select the Target Operating System—Windows or DOS.

3. If you want to change the current path, select the Drive and Directory to contain the file.

4. Enter a new File Name or accept the default executable name.
5. Choose OK.

   If a file having the same name already exists, you will be asked to confirm that you want to overwrite the file.

Note: If you create an executable to run under DOS, ensure the DCL commands in your script are supported under DOS. Refer to Chapter 8, “Command Reference” for a listing of commands not supported under DOS.

Distributing an Executable

When you distribute an executable to users in your organization to run under Windows, the following files must be in the same directory as the executable or on the path:

- ADSCON11.EBL
- ADSENV.DLL
- ADSPUB11.DLL
- ADSRUN11.DLL
- ADSUTILS.DLL
- CTL3D.DLL
- DCL.EBL
- WWNETWAR.DLL (For network functionality)
- WWNET.INI (For network functionality)
- NWCALLS.DLL (For network functionality)
- NWLOCALE.DLL (For network functionality)
- NWNET.DLL (For network functionality)

No support files are required for executables compiled to run under DOS.

Printing a Script

Use the following procedure to print all or part of the current script.

1. Choose File | Print.

   The Print dialog box is displayed.

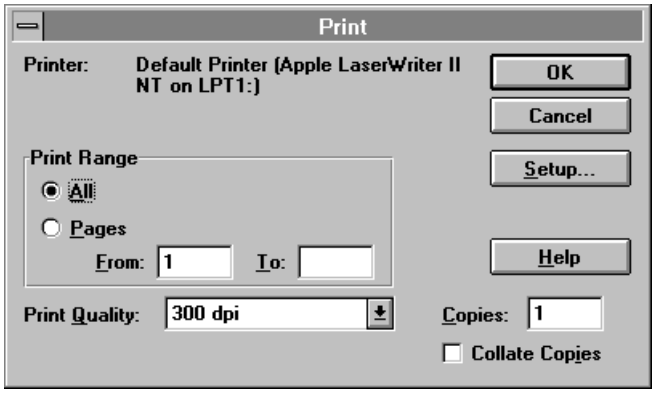


Figure 3-7: The Print dialog box

2. If you want to change the printer or printer settings, choose the Setup button and see “Setting Up for Printing” below.
3. Select the Print Range. Your choices are all of the file or a range of pages.
4. Change any of these print options as desired: Print Quality (depending on your printer), Copies, Collate Copies.
5. Choose OK.

Setting Up for Printing

Use the following procedure to set up your printer.

1. Choose File | Print Setup.

   The Print Setup dialog box is displayed.

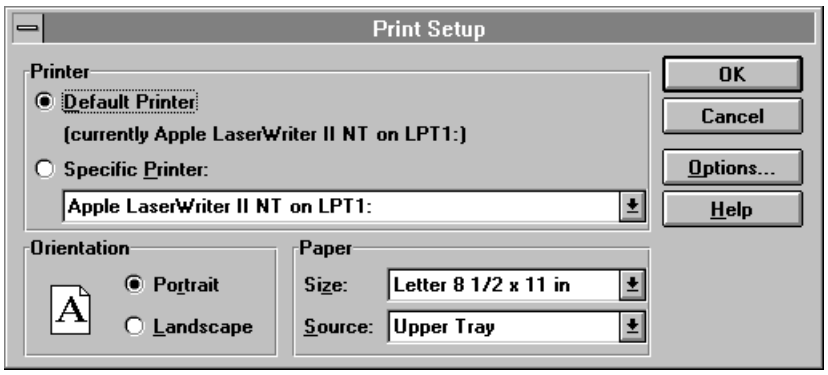


Figure 3-8: The Print Setup dialog box

2. Select the Default Printer or another Specific Printer.
3. Select the Orientation of the paper—portrait or landscape.
4. Select the desired Paper Size and Source, depending on your printer.
5. If you want to change the setting for your printer, choose the Options button. One or more dialog boxes specific to your printer will be displayed.
6. Choose OK.

Editing Text

The DCL Editor functions like other Windows text editors.

Use the following procedure to select a block of text for editing.

1. Position the insertion point at the beginning of the text to be selected and click the left mouse button.
2. Move the insertion point to the end of the block to be selected.
3. Hold down the Shift key and press the left mouse button.

OR

1. Move the insertion point to the beginning of the text to be selected.
2. Hold the Shift key.
3. Move the cursor to the end of the text to be selected.

   You can choose the Select All command from the Edit menu to select all of the text in the current document window.

Finding Text

Use the following procedure to find a string of text in a script.

1. Choose Edit | Find.

   The Find dialog box is displayed.

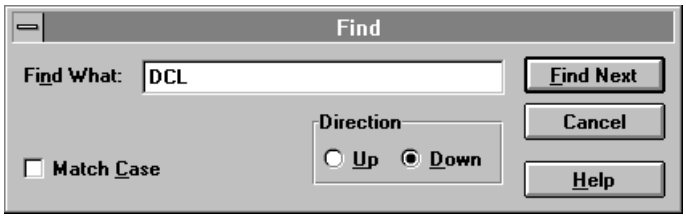


Figure 3-9: The Find dialog box

2. Enter the text string to search for in the Find What text box.

   Time Saver: If you select the text you want to search for before choosing the Find command, the selected text is put in the Find What text box for you.

3. Select the Match Case check box if you want the search to differentiate between upper- and lowercase letters.
4. Specify the Direction of the search—up or down.
5. Choose Find Next to begin the search.

   The first occurrence of the string after the insertion point is highlighted.

6. To continue searching for subsequent occurrences of the string, choose Find Next.

   The next occurrence of the string is highlighted.

   Note: You can edit the script the with the Find dialog box displayed. Use the mouse or press ALT+F6 to move between the dialog box and the script.

Replacing Text

Use the following procedure to replace one text string with another in the current script.

1. Choose Edit | Replace.

   The Replace dialog box is displayed.

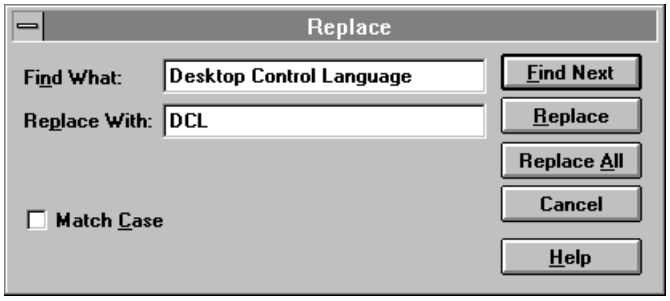


Figure 3-10: The Replace dialog box

2. Enter the text string to search for in the Find What text box.

Time Saver: If you select the text you want to search for before choosing the Replace command, the selected text is put in the Find What text box for you.

3. Enter the replacement text in the Replace With box.
4. Select the Match Case check box if you want the search-and-replace operation to differentiate between upper- and lowercase letters.
5. Choose Find Next to begin the search.

   The first occurrence of the string after the insertion point is highlighted.

6. If you want to substitute the replacement text for the string, choose Replace. (The search for the next matching string continues after the replacement is made.)

   OR

   If you want to go on to the next occurrence of the string without making the text change, choose Find Next.

   OR

   If you want to change all subsequent occurrences of the string to the replacement text, choose Replace All. (To avoid accidental changes, we recommend that you use this feature with caution.)

Running a Script

Use the following procedure to run a script that uses command-line arguments.

1. Choose Run | Arguments.

   The Set Arguments dialog box is displayed.

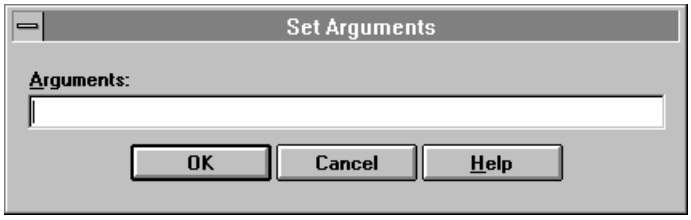


Figure 3-11: The Set Arguments dialog box

2. In the Set Arguments dialog box, enter the command-line arguments to be passed to the script's `main` subroutine and choose OK. (In your script, you can get these arguments through the `Command$` statement, which is described in Chapter 8, "Command Reference.")
3. Choose Run | Start Script.

   To stop the script that is currently executing, choose Run | End Script.

Running Executables

If the script has been saved as an executable file (using the File | Make Exe command), you can use the Applications Manager or Program Manager File | Run function or File Manager to run the script. For more information, see "Making an Executable" in this chapter.

Using DCL Tools

DCL provides a Debugger, Dialog Editor, Syntax Checker and Macro Recorder.

Starting the Debugger

To start the DCL Debugger, choose Debugger from the Tools menu. For details on how to use the Debugger, refer to Chapter 5 “Using the Debugger.”

Starting the Dialog Editor

To start the DCL Dialog Editor, choose Dialog Editor from the Tools menu. For details on how to use the Dialog Editor, refer to Chapter 4, “Using the Dialog Editor.”

Checking the Syntax

Use the following procedure to check the syntax of your script.

1. Position the insertion point (cursor) at the beginning of the script.
2. Choose Tools | Syntax Check.

   Upon encountering an error, the syntax checker stops at the line containing the error and displays an error message in a message box and in the Status Bar.
3. Correct the error.
4. Repeat Steps 1 – 3 until your script is free of syntax errors.

   If your script is free of errors, the message “Syntax OK” is displayed in the Status Bar.

Recording a Macro

DCL’s Recorder captures Windows events and translates them into DCL statements that can be inserted in a script. For a list of the of the events you can record, see “Recorded Events” in this chapter.

Use the following procedure to record Windows events.

1. Choose Tools | Recorder.

   The Script Recorder Options dialog box is displayed.

Figure 3-12: The Script Recorder Options dialog box

2. Make any desired changes to the dialog box:

   Include Comments: Indicate whether to include descriptive comments in the code generated by the Recorder.

   High-Level Statements: Indicate whether to consolidate multiple events into one High-Level Statement where possible. (For example, selecting a menu is recorded as a menu command rather than a series of mouse commands.)

   Keyboard: Indicate whether to capture events resulting from keyboard activity.

   Mouse: Indicate whether to capture events resulting from mouse activity then indicate whether the coordinate system is relative to the window or screen.

   Stop Recording On: Specify the key combination to stop the Recorder.

3. Choose OK.

   The DCL Editor window is minimized. The Recorder box is displayed.

Figure 3-13: The Recorder box

4. Perform the Windows events you want to record.

   You can interrupt recording by choosing Pause in the Recorder box. Then choose Record when you want to resume recording.

5. When you have finished recording events, choose End in the Recorder box.

   The DCL Editor window is restored; the Insert Recording dialog box is displayed.

Figure 3-14: The Insert Recording dialog box

6. Press F9 (or use mouse) to switch to your script.
7. Position the insertion point where you want to insert the recorded code.
8. Press F9 (or use mouse) to switch back to the Insert Recording dialog box.
9. Choose OK.

Recorded Events

The Recorder recognizes the Windows actions described in the following paragraphs. For further information about DCL commands, refer to Chapter 8, “Command Reference.”

Application Focus Switch

The Recorder recognizes when an application receives the focus and records this as a `WinActivate` statement if you perform an activity within the application. The focus can be shifted using any of the normal Windows application switching methods.

Window focus shifts are also recorded to ensure that subsequent actions occur (and are synchronized) within the appropriate window.

Statements

- `WinActivate`
- `AppActivate`

Window Scrolling

Interactions with windows that respond to scrolling messages (windows that have built-in Windows scroll bars) are recorded.

Statements

- `VLine`

- VPage
- HLine
- HPage
- HScroll
- VScroll

Mouse Activity

Mouse down and up actions are compressed to high-level forms when possible. Further, mouse activity that leads to a recognizable result, such as resizing a window, is compressed out completely.

Statements

- QueFlush
- QueMouseClick
- QueMouseDblClk
- QueMouseDblDn
- QueMouseDn
- QueMouseMove
- QueMouseUp
- QueSetRelativeWindow

Keyboard Activity

Keyboard press/release actions are converted. Complex key combinations, such as CTRL+ESC or ALT+SHIFT+C are converted into a readable form. Keyboard activity that results in a higher level command are compressed out completely, such as the following keystroke sequence, which results in a window reposition: ALT+Spacebar, M, Right, Right, Right, Enter.

Keyboard activity that modifies mouse events is also recorded, such as holding down the Shift key and clicking the left mouse button.

Statements

- `DoKeys`
- `QueFlush`
- `QueKeys`
- `QueKeyDn`
- `QueKeyUp`

Window Management

The Recorder recognizes high-level interactions with windows, such as movement, sizing, activation, minimizing, maximizing, and restoring. The Recorder differentiates between interactions with applications, popup windows and child MDI windows.

Statements

- `AppMaximize`
- `AppMinimize`
- `AppRestore`
- `AppMove`
- `AppSize`
- `WinMove`
- `WinSize`
- `WinMaximize`
- `WinMinimize`
- `WinRestore`

Menu Commands

The Recorder watches for interactions with an application's built-in menu and records the results of these interactions. All intermediate events used to select the menu choice (with the keyboard and mouse) are not recorded.

Statement

- `Menu`

Dialog Box Interaction

The Recorder recognizes interactions with standard Windows controls, such as edit boxes, check boxes, options buttons, list boxes and combo boxes. Also, interactions with known controls of other applications are supported, such as Borland's custom controls and the controls in Visual Basic.

Note: The results of dialog box interactions are recorded rather than the mouse/keyboard actions which resulted in those changes.

The Recorder also recognizes mixed dialogs—dialog boxes with a mixture of standard controls and custom controls. Many applications such as Control Panel, Corel Draw, and Lotus Ami Pro, use this technique. In this case, standard control interactions are recorded normally, while interactions with custom controls are recorded using high-level mouse and keyboard statements.

Statements

- `ActivateControl`
- `SelectComboBoxItem`
- `SelectButton`
- `SelectListboxItem`
- `SetCheckbox`
- `SetEditText`
- `SetOption`

Working with Document Windows

DCL Editor allows you to have several scripts open for editing at the same time. Each script is contained in a document window. Document windows can be moved, resized, maximized to fill the entire screen, minimized to an icon or restored.

The names of the currently open scripts are displayed at the bottom of the Window menu. To switch to another open script, choose the script name from the Window menu or click inside the window containing the script.

To display the document windows in a stair-step arrangement, choose Window | Cascade.

To display the document windows in a tiled arrangement, choose Window | Tile.

To neatly arrange the icons at the bottom of the DCL Editor window, choose Window | Arrange Icons.

Getting Help

DCL provides extensive Help that describes how to use the graphical script building environment and documents each DCL statement, function, system variable, etc.

To display the Table of Contents for the DCL Help, choose Help | Contents.

To search an index of DCL Help keywords, choose Help | Search for Help On.

For information on Microsoft Windows Help, choose Help | How to Use Help.

Context Sensitive Help

When you request context-sensitive Help, DCL displays a particular Help topic, based on your current activity. Context-sensitive Help is provided for each DCL language element, menu item, Toolbar button, and dialog box.

Use the following procedure to access help.

1. Press Shift + F1 or choose the  button from the Toolbar.

   The cursor changes to indicate that you are requesting context-sensitive Help.

2. Choose the item that you want help on.

   Help for the selected item is displayed.

Text Editing Keys

You can use the following keys when editing scripts:

Key	Description
Backspace	Deletes the selection or character preceding the cursor
Tab	Inserts a tab character
Enter	Inserts a new line, breaking the current line
CTRL+Insert, CTRL+C	Copies
SHIFT+Insert, CTRL+V	Pastes
SHIFT+Delete, CTRL+X	Cuts
Insert	Toggles between insert and overtype typing modes
Delete	Deletes the selection or character following the cursor
Up	Moves the cursor up one line
Down	Moves the cursor down one line
Left	Moves the cursor left one character position
Right	Moves the cursor right one character position
PgUp	Moves the cursor up by one window
PgDn	Moves the cursor down by one window
CTRL+PgUp	Scrolls the window left by one window
CTRL+PgDn	Scrolls the window right by one window
CTRL+Left	Moves the cursor left by one word
CTRL+Right	Moves the cursor right by one word
Home	Moves the cursor to the start of the line
End	Moves the cursor after the last character on the line
CTRL+Home	Moves the cursor to the first character in the macro
CTRL+End	Moves the cursor after the last character in the macro
SHIFT+Cursor Move	Drags the selection as the cursor moves
F1	Displays context-sensitive help for current command in editing window
SHIFT+F1	Displays context-sensitive help for current position.

Notes

Chapter 4 Using the Dialog Editor

This chapter describes how to use the Dialog Editor to interactively design dialog boxes to be used in DCL scripts. A mouse is required to use the Dialog Editor.

Starting the Dialog Editor

The procedure for starting the Dialog Editor differs, depending upon whether you are creating a new dialog box or modifying an existing one.

Use the following procedure to create a new dialog box template.

1. Position the insertion point where you want to include the new dialog box template.
2. Choose Tools | Dialog Editor.

   The Dialog Editor window containing a blank dialog box template is displayed.

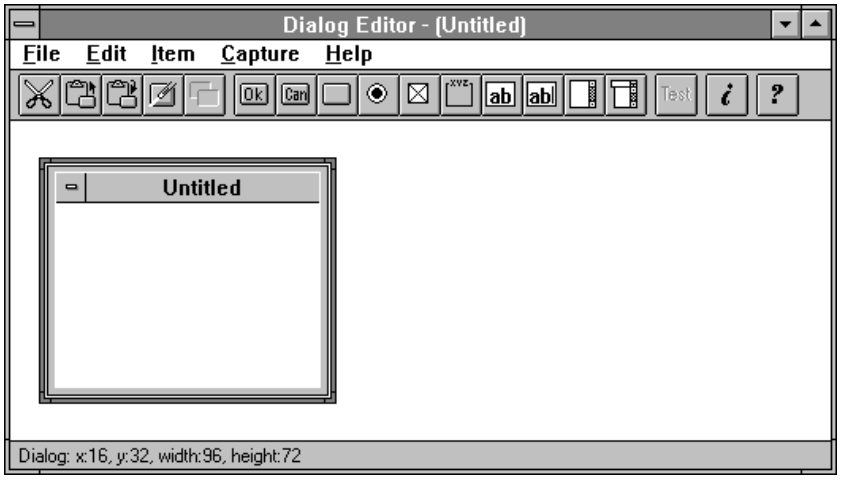


Figure 4-1: The Dialog Editor window

Use the following procedure to modify an existing dialog box template.

1. In the script, select the block of text containing the dialog box template.

   The block of text begins with a `Begin Dialog` statement and ends with an `End Dialog` statement. Be sure to include the `Begin Dialog` and `End Dialog` statements in the block of selected text.

2. Choose Tools | Dialog Editor.

   The text template is translated into a graphical representation in the Dialog Editor window.

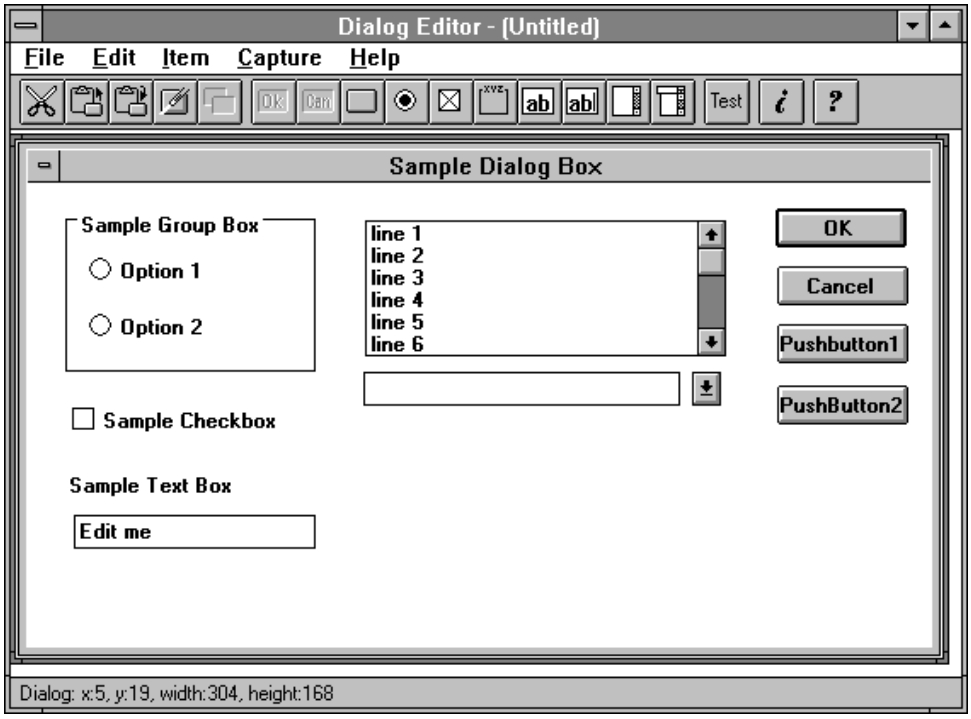


Figure 4-2: The Sample Dialog Editor window

Creating or Modifying a Dialog Box Template

The general process of creating or modifying a dialog box is:

- Start the Dialog Editor as described under “Starting the Dialog Editor” in this chapter. If you are creating a new dialog box template, an empty dialog box template is displayed.
- If you are modifying an existing dialog box template, the dialog box template contains the currently defined controls.
- Size the dialog box template by dragging the edges and corners.
- Add dialog box controls (buttons, options, boxes, text, etc.) to the dialog box template as described under “Adding Dialog Box Controls” in this chapter.
- Define the dialog box controls as described under “Defining Dialog Box Controls” in this chapter.
- Test the dialog box template as described under “Testing the Dialog Box Template” in this chapter.
- When you are satisfied with the dialog box template, choose Exit & Update from the File menu. The DCL statements defining the dialog box template are inserted in your script.

Note: For an example dialog box definition used in a script, see “Sample Script” in this chapter.

Using the Toolbar

The Toolbar is a row of buttons representing frequently used Dialog Editor commands. Instead of using the menus, you can choose a Toolbar button to execute a command.

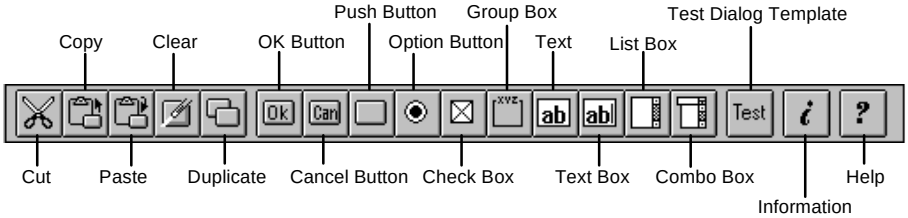


Figure 4-3: The Dialog Editor Toolbar

Using the Status Bar

The Status Bar appears at the bottom of the Dialog Editor window. It provides the name, coordinates and size of the selected control.

Figure 4-4: The Dialog Editor Status Bar

Adding Dialog Box Controls

You can add the following dialog box controls: OK button, Cancel button, Push button, Option (radio) button, Check box, Group box, Text, Text box, List box and Combo box.

Use the following procedure to add a dialog box control.

1. Select the control from the Item menu or Toolbar. This causes the cursor to take a shape resembling the control.
2. Move the cursor to the desired location in the dialog box template and click the left mouse button.
3. Size the control (by dragging edges and corners).

For all controls except the OK and Cancel buttons, you need to define the control as described under “Defining Dialog Box Controls” in this chapter.

For an example dialog box definition used in a script, see “Sample Script” in this chapter.

Defining Dialog Box Controls

After adding a dialog box control, you need to provide specific information to define the control. For example, after adding a Push button, you need to define the label to appear on the button.

Use the following procedure to define a dialog box control.

1. Select the control by clicking on it, using the left mouse button. A border appears around the selected control. To select the entire dialog box, click the title bar.
2. Choose Edit | Info.

   OR

   Click on the control again, this time using the right mouse button.

   OR

   Combine Steps 1 and 2 by double-clicking on the control.

   The <Dialog Box Control> Information dialog box is displayed. The title and contents of this dialog box vary depending on the control you are defining.

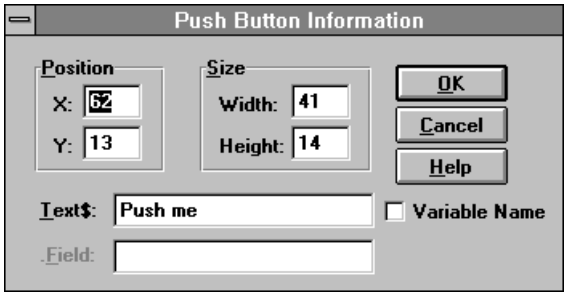


Figure 4-5: The Push Button Information dialog box

3. Use the Position and Size group boxes to change the upper left coordinates and size of the control if desired.

   (For an explanation of the unit of measurement used in dialog boxes, see the `Begin Dialog...End Dialog` command in Chapter 8, “Command Reference.”

4. Complete the Text$, Array$, and .Field items, depending on the type of control you are defining.

   For further information refer to, “Using Text$, Array$, and .Field Items” in this chapter.

5. Choose OK.

Using Text$, Array$, and .Field Items

This topic summarizes how to use the Text$, Array$, and .Field items in the <Dialog Box Control> Information dialog box. Each of these item is related to a DCL command.

For information regarding the DCL commands, refer to Chapter 8, “Command Reference.”

Text$ Note: If the text of a label is to be read from a string variable, check the Variable Name box on. In the **Dialog Editor** window, the name of the variable is displayed rather than literal text.

The Dialog Box

Use the Text$ field to enter a title for the dialog box.

Provide a Name for DCL to use to reference this dialog box template.

Related Command:

- `Begin Dialog`

OK and Cancel button

No action required. The `Dialog()` function, which displays the dialog in a running script, returns a value of -1 if the OK button was pressed or 0 for the Cancel button.

Related Commands:

- `OKButton`
- `CancelButton`

Push button

Use the Text$ field to enter the label for the button. The `Dialog()` function returns a value greater than 0 if a Push button was pressed. 1 is the value of the first Push button in the dialog box template, 2 is the second, etc.

Related Command:

- `PushButton`

Check box

Use the Text$ field to enter the label for the check box.

Provide a .Field variable name for this check box. It contains a -1 if the box is checked or a 0 if it is blank. This variable name must be unique within the dialog box template.

Related Command:

- `CheckBox`

Option group and buttons

An Option group is a collection of related Option buttons. For each Option button, use the Text$ field to enter the label for the option.

Use the .Field item to enter the same variable name for each Option button in the group. This integer variable indicates which option in the group is selected. A value of -1 indicates that no option is selected. 0 indicates that the first option in the dialog box template is selected, 1 indicates the second, etc.

Related Commands:

- `OptionGroup`
- `OptionButton`

List box

Use the Array$ item to provide the name of an array of strings to be displayed in the list box.

Use the .Field item to provide the name of an integer variable, which serves as the index to the array. This variable points to the selected array element. The first array element is 0, the second is 1, etc. When the array index is set to -1, no item is selected.

Related Command:

- `ListBox`

Combo box

Use the Array$ item to provide the name of an array of strings to be displayed in the list box.

Use the .Field item to provide the name of a variable that contains the string that occupies the combo box.

Related Command:

- `ComboBox`

Text

Use the Text$ field to enter the text to be displayed.

Related Command:

- `Text`

Text box

Provide a Field variable name for this text box. This string variable will contain the text for the text box. This variable name must be unique within the dialog box template.

Related Command:

- `TextBox`

Group box

Use the Text$ field to enter the label for the group.

Related Command:

- `GroupBox`

Cutting, Copying and Pasting Controls

The DCL Dialog Editor allows you to perform various graphical “editing” tasks.

Use the Cut command on the Edit menu to remove a control you have selected from the dialog box template and store it in the Windows clipboard.

Use the Copy command to copy the selected control to the clipboard.

Use the Paste command to insert the contents of the clipboard in the dialog box template.

Use the Clear command to remove the selected control from the dialog box template.

Use the Duplicate command to make a copy of the selected control in the dialog box template.

Testing the Dialog Box Template

To test the dialog box template, choose File | Test Dialog. Press F2 to return to the Dialog Editor.

Capturing Dialog Boxes

You can capture the controls from another dialog box to the Windows clipboard and then paste those controls in the Dialog Editor dialog box template.

Use the following procedure to capture controls from another dialog box.

1. If the dialog box you want to capture is visible with the Dialog Editor window displayed, choose Capture | Capture in Place.

   If the dialog box is hidden by the Dialog Editor window, choose Capture | Capture from Back. If you choose this command, the Dialog Editor window is hidden while you capture the dialog box.

   The mouse cursor takes the shape of a hook. When the cursor is over a capturable dialog, it looks like this:

2. Click on the dialog box you want to capture.

   The dialog box controls are stored in the Windows clipboard as DCL commands.

Return to the Dialog Editor window.

3. Click inside the dialog box template in the Dialog Editor window.
4. Choose Edit | Paste.

Opening an Existing File

You can save dialog box templates in text files. When you open the text file, the Dialog Editor translates the DCL commands into a graphical representation of the dialog box template.

Use the following procedure to open an existing file.

1. Choose File | Open.

   The Open Dialog File dialog box is displayed.

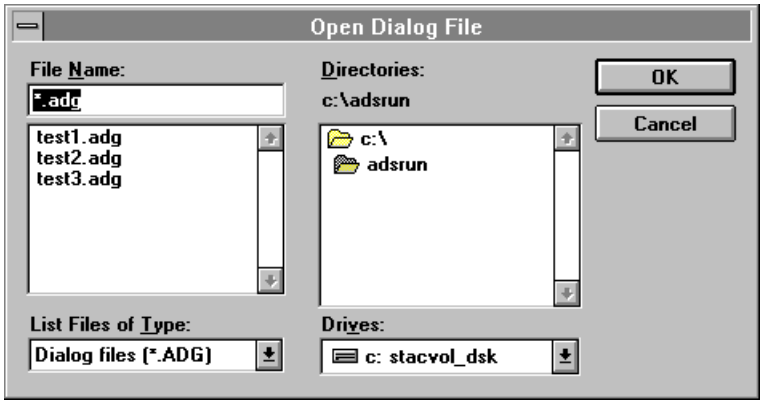


Figure 4-6: The Open Dialog File

2. Select the Drive and Directory containing the file.
3. Enter the File Name. Or select the file from the File Name list box.

   You can change the types of files displayed in the File Name list box by selecting another file type from the List Files of Type box.
4. Choose OK.

Saving a New Dialog Box Template

You can save the dialog box template to a text file. The Dialog Editor translates the graphical representation of the dialog box template into DCL commands.

Use the following procedure to save a new dialog box template

1. Choose File | Save As.

   The Save Dialog File box is displayed.

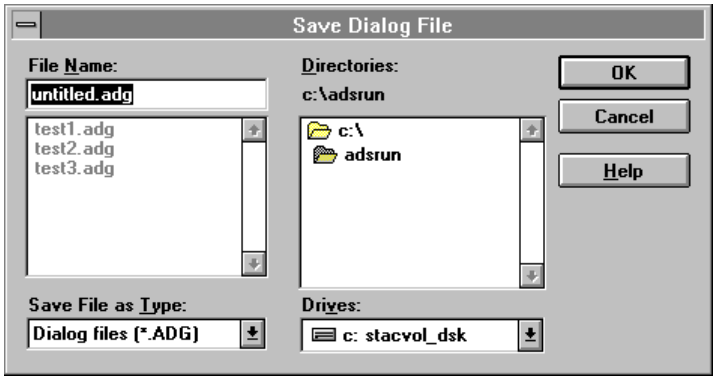


Figure 4-7: The Save Dialog File box

2. To change the current path, select the Drive and Directory to contain the file.
3. Enter the File Name.
4. Choose OK.

Saving an Existing File

Use the following procedure to save an existing file under a new name.

1. Choose File | Save As.

   The Save Dialog File dialog box is displayed.

2. Select the Drive and Directory to contain the file.
3. Enter the File Name.
4. From the Save File As Type box, select the desired file type.
5. Choose OK.

If a file having the same name already exists, you will be asked to confirm that you want to overwrite the file.

Running the Dialog Editor Standalone

You can also run the Dialog Editor as a standalone application, not as a part of the DCL Editor. When you run the Dialog Editor in this way, you must save your dialog box template as an .ADG file.

To start the Dialog Editor standalone, use the File | Run command in Applications Manager (or Program Manager) to start the program ADSDLG11.EXE in the default directory. Alternatively you can use File Manager to start the program.

Getting Help

DCL provides extensive Help that describes how to use the graphical dialog box building environment.

To display the Table of Contents for the Dialog Editor Help, choose Help | Contents.

To search an index of DCL Help keywords, choose Help | Search for Help On.

For information on Microsoft Windows Help, choose Help | How to Use Help.

Context Sensitive Help

The Dialog Editor displays a particular Help topic, based on your current activity when help is requested. Context-sensitive Help is provided for each menu item and Toolbar button.

Use the following procedure to access help.

1. Press Shift + F1.

   The cursor changes to indicate that you are requesting context-sensitive Help.

2. Choose the item you want Help on.

   The help for the selected item is displayed.

Sample Script

The following DCL script illustrates the definition and display of a dialog box as well as the interpretation of the user's input.

The main programming tasks required to implement a dialog box are: initialization, defining the dialog box template, displaying the dialog box and interpreting the user's interaction with the dialog box

The ↵ symbol indicates that the command should continue on the same line.

```
sub main()
    'Example showing the entire process of defining, displaying,
    'and interpreting the input from a dialog box.

    'Initialization
    '--------------
    'ListBox$ and ComboBox are single-dimensional arrays to hold the
    'contents of the list and combo box in the dialog box.
    Dim ListBox1$() as string
    Dim ComboBox1$() as string

    'Define dialog box
    '-----------------
    Begin Dialog UserDialog 16,32,304,168, "Sample Dialog Box"
        OKButton 251,9,44,14
        CancelButton 252,30,44,14
        PushButton 252,51,44,14, "Pushbutton1"
        PushButton 252,73,44,14, "PushButton2"
        GroupBox 13,9,84,59, "Sample Group Box"
        OptionGroup .OptionGroup1
            OptionButton 21,24,65,14, "Option 1"
            OptionButton 21,44,66,14, "Option 2"
        CheckBox 15,78,79,14, "Sample Checkbox", .CheckBox1
        Text 14,105,79,8, "Sample Text Box"
        TextBox 16,120,81,12, .TextBox1
        ListBox 114,14,120,48, ListBox1$, .ListBox1
        ComboBox 113,68,120,84, ComboBox1$, .ComboBox1
    End Dialog

    'Prepare to display dialog box.
    '------------------------------
    'Declare the dialog type using Dim ... as UserDialog command
    Dim aSampleDialog as UserDialog
```

```
    'Load the list box and combo box arrays with app window names
    AppList ComboBox1$
    AppList ListBox1$

    'Load the text box with an initial value
    aSampleDialog.TextBox1 = "12345678901234567890123456789O"
    'Note that you reference a dialog box field as follows:
    '   <DialogBoxName>.<Fiel dName>

    'Display the Dialog
    '------------------
    a% = Dialog(aSampleDialog)
                    'Returns integer indicating button chosen
    'Interpret user input
    '--------------------
    'This example builds a string describing the contents of the
    'dialog box when the user finished making changes.
    crlf$ = chr$(13)+chr$(10)
    dlgstr$ =  "Button pushed = "
    'Determine button pushed.
    select case a%
        case -1
            dlgstr$ = dlgstr$ + "OK"
        case 0
            dlgstr$ = dlgstr$ + "Cancel"
        case 1
            dlgstr$ = dlgstr$ + "PushButton1"
        case 2
            dlgstr$ = dlgstr$ + "PushButton2"
    end select
    dlgstr$ = dlgstr$ + crlf$
    'Determine new value of each dialog box component
    dlgstr$ = dlgstr$ +  "Option = " +   ↵
        str$(aSampleDialog.OptionGroup1) + crlf$
    dlgstr$ = dlgstr$ +  "Checkbox = " +   ↵
        str$(aSampleDialog.CheckBox1) + crlf$
    dlgstr$ = dlgstr$ +  "TextBox = " + aSampleDialog.TextBox1 + crlf$
    dlgstr$ = dlgstr$ +  "ListBox = " +   ↵
        ListBox1$(aSampleDialog.ListBox1) +   crlf$
    dlgstr$ = dlgstr$ +  "ComboBox = " + aSampleDialog.ComboBox1
    'Display the dlgstr$ string in a message box.
    msgbox dlgstr$
end sub
```

Exiting the Dialog Editor

When you have finished creating or modifying the dialog box template, choose File | Exit and Update. If you created a new dialog template, the dialog template commands will be added to your script. If you modified an existing dialog template from your script, the old template is replaced with the new one.

If you do not want to update your script, choose File | Exit. You will be asked whether to update your script before the connection to the Dialog Editor is terminated, choose no to exit.

Shortcut Keys

You can use the following keys when working with dialog box controls:

Key	Description
Ctrl+C	Copies the current control to the clipboard
Ctrl+V	Pastes the contents of the clipboard
Ctrl+X	Cuts the current control and store it in the clipboard
Ctrl+D	Duplicates the current control
Del	Clears the current control
Ctrl+I	Displays the Information dialog box
F1	Displays the Dialog Editor Help Contents
Shift+F1	Displays context-sensitive Help for menus and Toolbar buttons
F2	Toggles dialog box test on and off

Notes

Chapter 5 Using the Debugger

After you have checked the syntax of your script, the Debugger can help you find logic errors. The Debugger allows you to set breakpoints, establish watch variables, and trace through scripts statement by statement.

Starting the Debugger

Before starting the Debugger, choose the DCL Editor Tools | Syntax Check command to ensure that your script is free of syntax errors. The Debugger will not run or trace through a script with syntax errors.

To start the Debugger, choose the DCL Editor Tools | Debugger command.

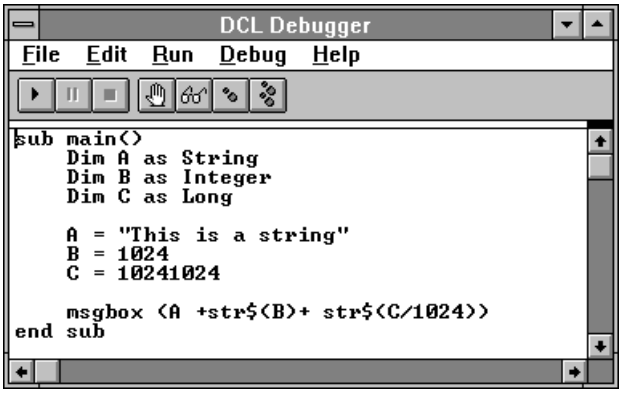


Figure 5-1: The Debugger window

Running a Script

You can run your script at full speed to reach the point where you want to begin debugging.

Before running the script, set a breakpoint (as described under “Setting Breakpoints” in this chapter) where you want to begin debugging. You can set breakpoints throughout your script.

To run a script at full speed, choose Run | Start. The script runs until it encounters the first breakpoint. (You must use the Run | Arguments command in the DCL Editor window to set command-line arguments, which are passed to `sub main`.)

At this point you can trace through the script as described under “Tracing Through a Script” in this chapter or continue at full speed to the next breakpoint by choosing Run | Continue.

To pause the execution of the script, choose Run | Break. To stop the script, choose Run | End.

Using the Toolbar

The Toolbar is a row of buttons representing frequently used **Debugger** commands. Instead of using the menus, you can choose a Toolbar button to execute a command.

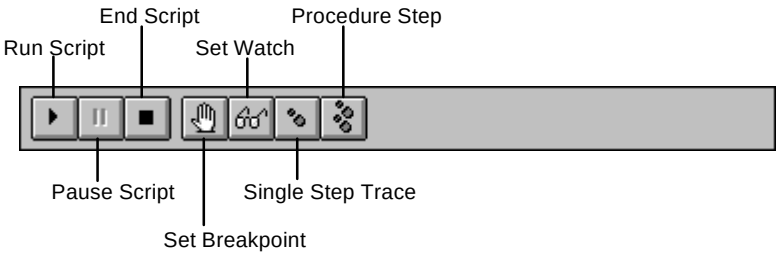


Figure 5-2: The Debugger Toolbar

Tracing Through a Script

Tracing means executing your script one line at a time. You can trace through your script from the beginning or run the script at full speed to the point where you want to begin debugging (as described under “Running a Script” in this chapter). While you are tracing through a script, the current line (called the *instruction pointer*) is highlighted.

To execute the current line and move the instruction pointer to the next line, choose Run | Single Step.

If you want to skip over a user-defined function or subroutine, choose Debug | Procedure Step. This command is the same as the Single Step command, except that it does not trace into user-defined subroutines or functions.

To reposition the instruction pointer, move the cursor to the line you want to make current and choose Debug | Set Next Statement.

Setting Breakpoints

To set a breakpoint, position the insertion point on the line where you want to set the breakpoint and choose Debug | Toggle Breakpoint. Do not set a breakpoint on a line that contains no code.

To toggle a breakpoint off, position the insertion point on the line where the breakpoint has been set and choose Debug | Toggle Breakpoint. To remove all breakpoints, choose Debug | Clear All Breakpoints.

You can set up to 255 breakpoints in a script.

All breakpoints are removed when you exit the Debugger.

Using Watch Variables

Watch variables appear in the watch pane of the Debugger window. As you step through your script, the current value of the variable and other information are displayed in the watch pane. If the variable is not currently in scope, the message <Not in Context> appears.

Opening the Watch Pane

When you start the DCL Debugger the watch pane is not open. The *splitter* (pictured in the following window) for the closed pane appears just below the Toolbar.

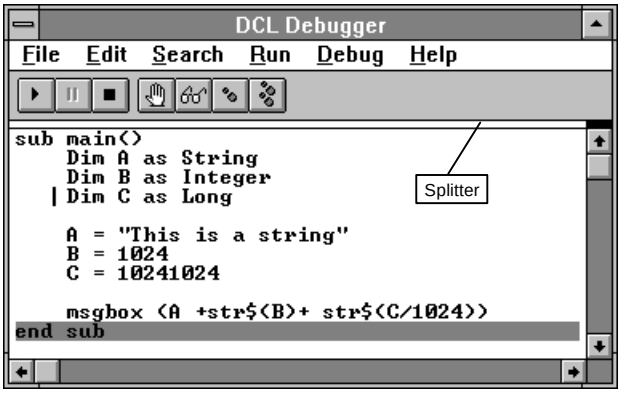


Figure 5-3: Window with the watch pane closed

To open the *watch pane* (pictured in the following window), click inside the splitter, hold the mouse button down and drag the pane to the desired size.

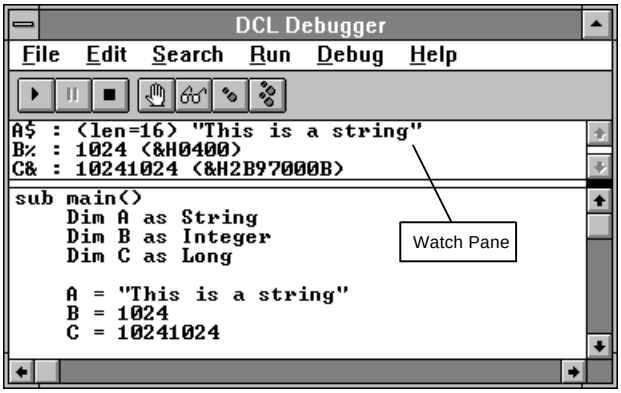


Figure 5-4: Window with the watch pane open

For a variable to appear in the watch pane, you must explicitly add it.

Use the following procedure to add a watch variable.

1. Choose Debug | Add Watch.

   The Add Watch dialog box is displayed.

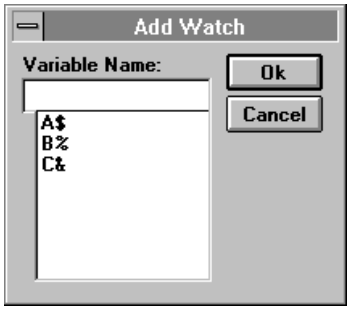


Figure 5-5: The Add Watch dialog box

2. Enter the name of the variable you want to watch or select it from the list box.

   The variable will not show up in the Add Watch dialog box until you have begun to step through the script.

3. Choose OK.

   Once added the variable appears in the watch pane.

Deleting a Watch Variable

Use the following procedure to delete a variable from the Watch Pane.

1. Select the variable in the watch pane.
2. Choose Debug | Delete Watch or press the DEL key.

Editing Text

The Debugger provides the basic text editing functions.

Use the following procedure to select a block of text for an editing operation.

1. Position the insertion point at the beginning of the text to be selected and click the left mouse button.
2. Move the insertion point to the end of the block to be selected.
3. Hold down the Shift key and press the left mouse button.

OR

1. Move the insertion point to the beginning of the text to be selected.
2. Hold the Shift key.
3. Move the cursor to the end of the text to be selected.

Finding Text

Use the Find command to search for a particular text string in the script.

Use the following procedure to find a string of text in a script.

1. Choose Search | Find.

   The Find dialog box is displayed.

Figure 5-6: The Find dialog box

2. Enter the text string to search for in the Find What text box.
3. Select the Match Case check box if you want the search to differentiate between upper- and lowercase letters.
4. Specify the Direction of the search—up or down.
5. Choose Find Next to begin the search.

   The first occurrence of the string after the insertion point is highlighted.

6. To continue searching for subsequent occurrences of the string, choose Find Next.

   The next occurrence of the string is highlighted.

Note: You can also continue the search by closing the dialog box (by choosing Cancel) and choosing Find Next from the Search menu.

Replacing Text

Use the Replace command to replace one text string with another in the current script.

Use the following procedure to replace one text string with another.

1. Choose Search | Replace.

   The Replace dialog box is displayed.

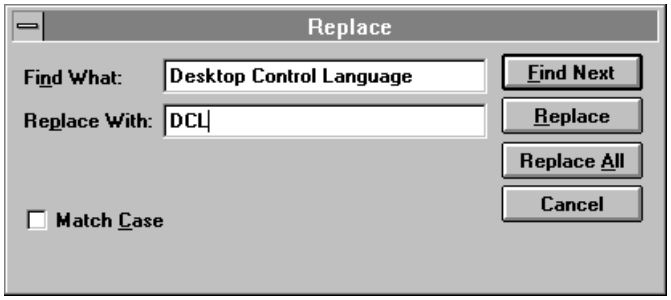


Figure 5-7: The Replace dialog box

2. Enter the text string to search for in the Find What text box.
3. Enter the replacement text in the Replace With box.
4. Select the Match Case check box if you want the search-and-replace operation to differentiate between upper- and lowercase letters.
5. Choose Find Next to begin the search.

   The first occurrence of the string after the insertion point is highlighted.

6. If you want to substitute the replacement text for the string, choose Replace. (The search for the next matching string continues after the replacement is made.)

   OR

   If you want to go on to the next occurrence of the string without making the text change, choose Find Next.

   OR

   If you want to change all occurrences of the string to the replacement text, choose Replace All.

Editing Dialogs

You can use the Dialog Editor (described in Chapter 4) to insert a new dialog box template in the script you are debugging or update an existing dialog box template from your script.

Use the following procedure to create a new dialog box template.

1. Choose Edit | Insert New Dialog.
2. Use the Dialog Editor to create the dialog box template.
3. Choose File | Exit and Update to exit the Dialog Editor and insert the template in the script.

Use the following procedure to update a dialog box template.

1. Select the text containing the dialog box template.
2. Choose Edit | Edit Dialog.
3. Use the Dialog Editor to modify the dialog box template.
4. Choose File | Exit and Update to exit the Dialog Editor and replace the old template in the script.

Getting Help

DCL provides extensive Help that describes how to use the Debugger.

To display the Table of Contents for the Debugger Help, choose Help | Contents.

To search an index of DCL Help keywords, choose Help | Search for Help On.

For information on Microsoft Windows Help, choose Help | How to Use Help.

Context Sensitive Help

When you request context-sensitive Help, DCL displays a particular Help topic, based on your current activity.

For Help on a particular DCL language element (statement, function, or system variable) enter the language element in the document and press F1.

For help on a menu item or Toolbar button:

1. Press Shift + F1.

   The cursor changes to indicate that you are requesting context-sensitive Help.

2. Choose the item you want Help on.

   Help for the selected item is displayed.

Exiting the Debugger

When you have finished debugging your script, choose File | Exit and Update. Your script will be updated with any changes you made during the debugging process.

If you do not want to update your script, choose File | Exit. You will be asked whether to update your script before the Debugger is closed choose No to exit.

Shortcut Keys

You can use the following keys when debugging scripts:

Key	Description
ESC	Closes the Debugger
CTRL+C or CTRL+Insert	Copies
Ctrl+V or SHIFT+Insert	Pastes
Ctrl+X or SHIFT+Delete	Cuts
DEL	Clears
	OR
	Deletes the current Watch variable if the Watch window is active
F3	Finds Next

Key	Description
SHIFT+F8	Procedure Step
F8	Single Step
F9	Toggles Breakpoint on or off
F6	Moves between the Watch window and the editing workspace if the Watch window is open
F1	Displays context-sensitive Help for current command in editing window
SHIFT+F1	Displays context-sensitive Help for menus and Toolbar buttons

Chapter 6 DCL Characteristics

This chapter provides an explanation of the language characteristics of DCL.

General Characteristics

- DCL is compiled, not interpreted.
- A DCL script consists of any number of user-defined functions, user-defined subroutines, and external DLL declarations.
- The compiler and runtime operate under Windows 3.1 and Windows for Workgroups 3.11 or later.
- A DCL script is an ASCII stream of text with a NULL terminator.
- DCL scripts are completely self contained. Variables have a scope local to the function or subroutine in which they are declared. Further, any referenced functions or subroutines must occur within the same script.
- Line numbers are not supported. Labels are used instead.
- Quotes can be embedded within constant string expressions using two consecutive quotes, such as `"John said, ""hello"", to Jane"` .
- Numeric constant folding is supported. For example, the statement `i=10+23` is reduced before compiling to `i=33`.
- Constant expressions involving strings are reduced at compile time, including calls to the function `CHR$()` where the passed parameter is a constant. For example, the statement `s$ = "Hello" + chr$(13) + chr$(10) + "world"` is reduced to the expression `s$ = "Hello\r\nworld"` .
- The runtime is reentrant. Thus, two DCL scripts can execute concurrently.

Script Execution

Execution of a DCL script begins with a subroutine with the predefined name `Main`.

Structures

User-defined structures are not supported. DCL does support dialog structures.

Compiler

- All built-in statements, functions, and constants are implicitly declared and compiled in a resource file. Thus, there are no include files required for compiling.
- Compiler metacommands are not supported.
- The compiler is reentrant. Thus, two DCL scripts can be compiled simultaneously.

Variables

- DCL supports integers, longs, shorts, strings, dialogs, single precision floats and double precision floats. Refer to “Data Types” in this chapter for more information.
- Variables declared (explicitly or implicitly) within a subroutine are local to that subroutine.
- Variables that are not explicitly given a type (with the `dim` statement or using a type specifier) are given the type `integer`.
- All declared variables are assigned an initial value of 0. For strings, 0 equates to "" (or NULL).
- Internal type conversions are performed automatically between any two numeric quantities. Thus, the script author can freely assign numeric quantities without regard to type conversions. However, it is possible for an overflow error to occur when converting from larger to smaller types. This occurs when the larger type contains a numeric quantity that cannot be represented by the smaller type.

For example, the following code will produce a runtime error:

```
dim amount as long
dim quantity as integer
amount = 400123      'assign a value out of range for int
quantity = amount    'attempt to assign to integer
```

Like many runtime errors, the overflow error is trappable.

- Loss of precision is not a runtime error.
- The declaration modifiers `STATIC`, `GLOBAL`, and `SHARED` are not supported.

Expression Evaluation

DCL allows expressions to involve data of different types. When this occurs, the two arguments are converted to be of the same type by promoting the "smaller" of the two data types. For example, DCL will promote the value of i% to a double in the following expression:

```
result# = i% * d#
```

When evaluating an expression, DCL also looks at the resultant type. During evaluation, the data type of each operand of each subexpression is compared with the data type of the result. Each operand will then be promoted to be the same as the "largest" of the two operands and the result. For example, the following expression results in the value `2.5`, even though the division appears to involve two integer values:

```
d# = 10 / 4 'd# becomes 2.5
```

The expression evaluator realizes that the result of the expression is being assigned to a `double`, and promotes the two integer operands of the division to be of the same type as the result. After the promotion, the division is performed. If this promotion did not occur, then the above expression would result in `2`, which would be incorrect.

Subroutines & Functions

- Subroutines and functions are available to any other subroutine or function within the same script.
- Arguments to functions and subroutines are passed automatically by reference. Passing by value is accomplished with the BYVAL keyword:

```
sub foo(byval a as integer)
    :
end sub
    sub main
    d% = 10
    food        'pass d by value
end sub
```

If a subroutine declares parameters by reference, then the caller can force a parameter to be passed by value by enclosing the parameter with parentheses:

```
sub foo(a as integer)
    :
end sub
    sub main
    d% = 10
    foo(d)  'force 'd' to be passed by value
end sub
```

- Forward references are not allowed. The body of a referenced subroutine or function must appear before its reference in the script.
- Recursive statements and functions are supported.

Error Handling

- DCL supports error handling in a manner that conforms to the Visual Basic error handling model. Individual error numbers may be different.
- ON ERROR traps are valid only during the execution of the current subroutine or function.
- Error traps are saved and restored within a user-defined subroutine or function.

DCL errors fall into three categories:

- Trappable error numbers are between 10 and 1000.
- Non-trappable error numbers are between 0 and 10. Internal errors and out of memory errors are all non-trappable.
- Application specific error numbers are greater than or equal to 1000.

DCL uses the Windows floating point emulator WIN87EM.DLL. Floating point exception errors are handled in a standard way using the SDK `__fpmath()` function. When the host application calls the compiler or the runtime, the floating point exception handler of the host application is saved and restored on exit.

Arrays

- Dynamic allocation of arrays is supported using both the `DIM` and the `REDIM` statements.
- Arrays can be declared to contain any fundamental data type.
- Arrays can have up to 60 dimensions.
- Array dimensions and size can be changed dynamically using the `REDIM` statement.
- Arrays are always passed by reference.

Data Types

Integer	Type Declaration	%
	Significant Digits	4
	Size	2 bytes (16 bits)
	Range	-32768 <= X <= 32767
Long	Type Declaration	&
	Significant Digits	9
	Size	4 bytes (32 bits)
	Range	-2147483648 <= X <= 2147483647

String	Type Declaration	$
	Significant Digits	N/A
	Size	1 byte per character
	Range	0 <= LENGTH < 32768
	Note	All strings are variable length. Fixed length strings are not yet supported.
Single	Type Declaration	!
	Significant Digits	7
	Size	4 bytes (32 bits)
	Range	Negative Values: -3.402823E38 to -1.401298E-45 Positive Values: 1.401298E-45 to 3.402823E38
Double	Type Declaration	#
	Significant Digits	15-16
	Size	8 bytes (64 bits)
	Range	Negative Values: 1.797693134862315E308 to -4.94066E-324 Positive Values: 4.94066E-324 to 1.797693134862315E308

Operator Precedence

The following table lists the operators and their descriptions in order of precedence.

Operators	Description
()	parentheses
^	exponentiation
-	unary minus
/, *	multiplication and division
\	integer division
mod	module
+, -	addition and subtraction
=, <>, >, <, <=, >=	relational
not	logical negation
and	and
or	or
xor	exclusive or

DCL Comments

Comments can be added to DCL code in the following manner:

- All text between a single quote and the end of the line is ignored:

```
MsgBox "hello"  'display a message box
```

- The `REM` statement causes the compiler to ignore the entire line:

```
REM This is a comment...
```

- DCL supports C-style multi-line comment blocks `/*....*/`, as shown in the following example:

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

  The C-style comments cannot be nested.

Constants

- Numerous symbolic constants that you can use with specific DCL commands are defined in the files ADSCON11.EBL and DCL.EBL. The constants defined for a particular DCL command are listed in Chapter 8, “Command Reference.” Two constants are defined by DCL itself—TRUE and FALSE.
- The following constants are predefined for use with DCL:

ATTR_ARCHIVE	NS_LOGGEDON
ATTR_DIRECTORY	PO_LANDSCAPE
ATTR_HIDDEN	PO_PORTRAIT
ATTR_NONE	TRUE
ATTR_NORMAL	TYPE_DOS
ATTR_READONLY	TYPE_WINDOWS
ATTR_SYSTEM	VK_LBUTTON
ATTR_VOLUME	VK_RBUTTON
ENV_BOTH	VS_VERSION_INFO
ENV_DOS	WS_MAXIMIZED
ENV_WINDOWS	WS_MINIMIZED
FALSE	WS_RESTORED
NS_ACTIVE	

DCL Limitations

- Each running script is limited to 64K of data. The data segment contains dynamic variables (such as strings and arrays), constants, an event jump table, and external DLL call information.
- Strings are limited to 32K in size (32767 bytes).
- Script source size is limited to 42K.
- Compiled code size is limited to 64K. The code segment contains the executable pcode.
- The maximum size of the symbol table (used by runtime errors and the debugger) is 64K.
- Arrays can have up to 60 dimensions.
- Variable names are limited to 40 characters.
- Labels are limited to 40 characters.
- A given subroutine or function can have up to 100 variables, including variables that are passed.
- The stack size for the runtime is 2048 bytes.
- The number of open DDE channels is unlimited (limited only by available memory and system resources).
- The number of open files is limited to 255, or the operating system limit, whichever is less.
- The number of characters within a string literal (a string enclosed within quotes) is 255 characters. (Strings can be concatenated using the plus (+) operator with the normal string limit of 32767 characters.)
- Number of nesting levels is limited by compiler memory.
- Text file input/output buffer size is 512 bytes.
- Queue playback buffer size is limited to 64K. With 10 bytes per event, this allows for 6553 events. This memory is buffered in blocks of 100 events (1000 bytes).
- Each `gosub` requires 2 bytes of the DCL runtime stack.

Notes

Chapter 7 Related Commands

This chapter categorizes the DCL language elements into groups that perform related tasks.

Arrays

Description	Function/Statement
Change default lower limit	`Option Base`
Declare and initialize	`Dim`
	`FileDirs`
	`FileList`
	`ReDim`
	`ReadINISection`
Find the limits	`LBound` `UBound`
Manipulate an array	`ArrayDims` `ArraySort`

Clipboard Manipulation

Description	Function/Statement
Capture a screen or window to the clipboard	`Snapshot`
Clear the clipboard object	`ClipboardClear`
Get contents of clipboard	`Clipboard$`

Conversions

Description	Function/Statement
ANSI value to string	`Chr$`
Date to serial number	`DateSerial` `DateValue`

Number to string	`Cstr`
	`Str$`
	`Oct$`
	`Hex$`

One numeric type to another	`CDbl`
	`CInt`
	`CLng`
	`CSng`

String to ASCII value	`Asc`
String to number	`Val`

Date and Time Functions

Description	Function/Statement
Date to serial number	`DateSerial` `DateValue`

Get the current date or time	`Date$`
	`Now`
	`Time$`

Serial number to date	`Day`
	`Month`
	`Weekday`
	`Year`

Serial number to time	`Hour`
	`Minute`
	`Second`

Set the date or time	`Date$` `Time$`

Time a process	Timer
Time to serial number	TimeSerial TimeValue

DCL Environment Information

DCLHomeDir$

DCLOS$

DCLVersion$

Desktop Modifications

DesktopCascade

DesktopSetColors

DesktopSetWallpaper

DesktopTile

Dialog Creation

Begin Dialog...EndDialog

CancelButton

Checkbox

Combobox

Dialog

Dim

GroupBox

ListBox

OKButton

OptionButton

OptionGroup

PushButton
Text
TextBox

Dialog Display

AnswerBox
AskBox$
AskPassword$
InputBox$
MsgBox
MsgClose
MsgOpen
MsgSetText
MsgSetThermometer
OpenFileName$
PopupMenu
SaveFileName$
SelectBox

Dialog Manipulation

ActivateControl
ButtonEnabled
ButtonExists
CheckboxEnabled
CheckboxExists
ComboboxEnabled
ComboboxExists
EditEnabled
EditExists

GetCheckbox
GetComboboxItem$
GetComboboxItemCount
GetEditText$
GetListboxItem$
GetListboxItemCount
GetOption
ListboxEnabled
ListboxExists
OptionEnabled
OptionExists
SelectButton
SelectComboboxItem
SelectListboxItem
SetCheckbox
SetEditText
SetOption

Dynamic Data Exchange (DDE)

DDEExecute
DDEInitiate
DDEPoke
DDERequest
DDETerminate
DDETerminateAll
DDETimeOut

Environment Statements and Functions

Description	Function/Statement
Control an MCI device	MCI
End the working session	SystemRestart
Get information about the system	ReadINI$ ReadINISection SystemFreeMemory SystemFreeResources SystemTotalMemory SystemWindowsDirectory$ SystemWindowsVersion$
Modify the Windows environment	AddIni RefreshIni SystemMouseTrails WriteINI
Save and restore the environment	RestoreEnv SaveEnv
Set and get environment variables	Environ$ GetEnv SetEnv

Error Trapping

Description	Function/Statement
Get error messages	Error$
Get error-status data	Erl
Get or set error-status data	Err
Simulate run-time errors	Error
Trap errors while a program is running	On Error Resume

File Input and Output

Description	Function/Statement
Access or create a file	Open
Close files	Close Reset
Copy a file	FileCopy
Get information about a file	EOF FileAttr FileDateTime FileExists FileLen FileList FileType FindFile$ FreeFile Loc LOF Seek
Manage disk drives or directories	ChDir ChDrive CurDir$ DiskDrives DiskFree MkDir RmDir DirExists
Manage files	Dir$ FileDirs FileParse Kill Name
Read from a file	Input # Input$ Line Input #
Set or get file attributes	GetAttr SetAttr
Set read-write position in a file	Seek
Write to a file	Print # Write #

Flow Control

Description	Function/Statement

Branch	`GoSub...Return`
	`Goto`
	`On Error`

Exit or pause the program	`EnableStopScript`
	`End`
	`SleepUntil`
	`Stop`
	`WaitForTaskCompletion`

Loop	`Do...Loop`
	`Exit Do`
	`Exit For`
	`For...Next`
	`While...Wend`

Make decisions	`If...Then...Else` `Select Case`
Pause script	`Sleep`
Prevent other applications	`Exclusive`
Run another program	`Shell`
Yield control to other applications	`DoEvents`

Icons

`SetIcon`

`SetIconTitle`

`ShowIcon`

Keyboard Manipulation

Description	Function/Statement
Play back a string of keystrokes	DoKeys
Record and play keystrokes, using the event queue	QueEmpty QueFlush QueKeyDn QueKeys QueKeyUp SendKeys

Math Statements and Functions

Description	Function/Statement
General calculations	Exp Log Sqr
Generate random numbers	Random Randomize Rnd
Get absolute value	Abs
Get the sign of an expression	Sgn
Numeric conversion	Fix Int
Trigonometry	Atn Cos Sin Tan

Menus

Description	Function/Statement
Issue menu command	`Menu`

Manipulate menu items	`MenuItemChecked`
	`MenuItemEnabled`
	`MenuItemExists`

Miscellaneous Statements and Functions

Description	Function/Statement
Comment	`Rem` `'`
Get command-line arguments	`Command$`
Sound a beep	`Beep`

Mouse Events

`QueEmpty`

`QueFlush`

`QueMouseClick`

`QueMouseDblClk`

`QueMouseDblDn`

`QueMouseDn`

`QueMouseMove`

`QueMouseUp`

`QueSetRelativeWindow`

Network Functions

NetAttach
NetConnectDrive
NetDetach
NetDirectoryRights
NetDisconnectDrive
NetGetDirectoryRights
NetMemberOf
NetStationID
NetUserName
NetworkStatus

Operators

Description	Function/Statement
Arithmetic	*
	+
	-
	/
	\
	^
	Mod
Comparison	<
	<=
	<>
	=
	>
	>=
Logical	And
	Not
	Or
	Xor

Printer Manipulation

PrinterGetOrientation

PrinterSetOrientation

PrintFile

Procedure Statements

Description	Function/Statement
Call a subroutine	Call
Declare a reference to an external routine	Declare
Define a procedure	Function...End Function Main Sub...End Sub
Exit a procedure	Exit Function Exit Sub

Strings

Description	Function/Statement
Convert to lowercase or uppercase letters	LCase$ UCase$
Convert string to number	Val
Create strings of repeating characters	Space$ String$

Description	Function/Statement
Find the length of a string	Len

Logical operators used in string comparisons	<
	<=
	<>
	=
	>
	>=
Manipulate strings	InStr
	Left$
	LTrim$
	Mid$
	Null
	Right$
	RTrim$
	StrComp
	StringSort
	Str$
	Trim$
Parsing	Item$
	ItemCount
	Line$
	LineCount
	Word$
	WordCount
Work with ASCII and ANSI values	Asc
	Chr$

Variables and Constants

Description	Function/Statement
Assign value	`Let`

Declare variables or constants	`Const`
	`Dim`
	`ReDim`

Set default data type	`Deftype`

Viewport Window Manipulation

`Print`

`ViewportClear`

`ViewportClose`

`ViewportOpen`

Window Manipulation

`AppActivate`

`AppClose`

`AppFileName$`

`AppFind`

`AppGetActive$`

`AppGetPosition`

`AppGetState`

`AppHide`

`AppList`

`AppMaximize`

`AppMinimize`

`AppMove`

`AppRestore`

AppSetState
AppShow
AppSize
AppType
HLine
HPage
HScroll
VLine
VPage
VScroll
WinActivate
WinClose
WinFind
WinList
WinMaximize
WinMinimize
WinMove
WinRestore
WinSize

Notes

Chapter 8 Command Reference

This chapter describes each element (statement, function, constant, operator, etc.) provided by the Desktop Control Language (DCL). The language elements are listed alphabetically. For most elements, the following information is provided:

- Description.
- Syntax.
- Comments offering further explanation.
- Example. This is either a code sample or a reference to a topic containing a code sample.
- See Also. This is a reference to groups of related language elements in Chapter 7, “Related Commands” and to additional elements in this chapter.

The following symbols are used in this chapter:

Indicates that a statement or function cannot be used in a DOS environment. (It can only used under Windows.)

A single DCL command cannot be broken into multiple lines separated by carriage return/line feeds ((Chr$(13)+Chr$(10)). Because of the width of the physical page, long commands used in code samples must be broken. In these cases, the ↵ symbol indicates that the current line continues without a carriage return/line feed.

'

Description Begins a comment line or comments the remainer of the current line.

Syntax `' text`

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

See Also Comments (Chapter 6)

REM Statement

*

Description Multiplication operator.

Syntax `expression1 * expression2`

Returns Returns the product of `expression1` and `expression2`.

See Also Operators (Chapter 7)

+

Description Addition operator.

Syntax `expression1 + expression2`

Returns Returns the sum of `expression1` and `expression2`.

See Also Operators (Chapter 7)

-

Description Subtraction operator.

Syntax `expression1 - expression2`

Returns Returns the difference between `expression1` and `expression2`.

See Also Operators (Chapter 7)

/

Description Long Division operator.

Syntax `expression1 / expression2`

Returns Returns the quotient of `expression1` and `expression2`.

See Also Operators (Chapter 7)

<

Description Less Than operator.

Syntax `expression1 < expression2`

Returns TRUE if `expression1` is less than `expression2`, otherwise FALSE. If the two expressions are strings, then the operator performs a case-sensitive comparison between the two expressions, returning TRUE if `expression1` is less than `expression2`.

See Also Operators (Chapter 7)

Strings (Chapter 7)

<=

Description Less Than or Equal To operator.

Syntax `expression1 <= expression2`

Returns TRUE if `expression1` is less than or equal to `expression2`, otherwise FALSE. If the two expressions are strings, then the operator performs a case-sensitive comparison between the two expressions, returning TRUE if `expression1` is less than or equal to `expression2`.

See Also Operators (Chapter 7)

Strings (Chapter 7)

<>

Description Not Equal operator.

Syntax `expression1 <> expression2`

Returns TRUE if `expression1` is not equal to `expression2`, otherwise FALSE. If the two expressions are strings, then the operator performs a case-sensitive comparison between the two expressions, returning TRUE if `expression1` is not equal to `expression2`.

See Also Operators (Chapter 7)

Strings (Chapter 7)

=

Description Equal To operator.

Syntax `expression1 = expression2`

Returns TRUE if `expression1` is equal to `expression2`, otherwise FALSE. If the two expressions are strings, then the operator performs a case-sensitive comparison between the two expressions, returning TRUE if `expression1` is equal `expression2`.

See Also Operators (Chapter 7)

Strings (Chapter 7)

>

Description Greater Than operator.

Syntax `expression1 > expression2`

Returns TRUE if `expression1` is greater than `expression2`, otherwise FALSE. If the two expressions are strings, then the operator performs a case-sensitive comparison between the two expressions, returning TRUE if `expression1` is greater than `expression2`.

See Also Operators (Chapter 7)

Strings (Chapter 7)

>=

Description Greater Than or Equal To operator.

Syntax `expression1 >= expression2`

Returns TRUE if `expression1` is greater than or equal to `expression2`, otherwise FALSE. If the two expressions are strings, then the operator performs a case-sensitive comparison between the two expressions, returning TRUE if `expression1` is greater than or equal to `expression2`.

See Also Operators (Chapter 7)

Strings (Chapter 7)

\

Description Integer Division operator.

Syntax `expression1 \ expression2`

Returns Returns the integer portion of the quotient of `expression1` divided by `expression2`. Any fractional part of the answer is dropped. For example 3 \ 1 equals 1 rather than 1.5.

See Also Operators (Chapter 7)

^

Description Exponentiation operator.

Syntax `expression1 ^ expression2`

Returns Returns `expression1` raised to the power specified by `expression2`.

See Also Operators (Chapter 7)

Abs Function

Description Returns the absolute value of a given number.

Syntax `abs(number)`

Example

```
sub main()
   'ABS() - absolute value
   dim a as single
   dim b as integer

   a = 2.34
   b = abs(a)
   msgbox str$(b)
end sub
```

See Also Math Statements and Functions (Chapter 7)

ActivateControl Statement

Description Sets the focus to the control with the specified name or ID.

Syntax 1 `ActivateControl ControName$`

Syntax 2 `ActivateControl ControlID%`

Comments The control can be referenced using either the `ControlName$` or the `ControlD%`.

For push buttons, radio buttons, or check boxes, `ControlName$` is the name of the actual button. For list boxes, combo boxes, and edit boxes, `ControlName$` is the name that appears within the static text control that immediately precedes it in the window manager list.

A runtime error is generated if a control with `ControlName$` or `ControlID%` cannot be found.

This statement is used primarily to set the focus to a custom control within a dialog box. This is accomplished by setting the focus first to a known control that immediately precedes the custom control, then simulating a TAB key press:

```
ActivateControl "Portrait"
DoKeys "{TAB}"
```

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

AddIni Function

Description Reads the .INI settings of the source .INI file and adds them to the destination .INI file.

Syntax `AddIni(srcfile$[, destfile$])`

Comments If the `destfile$` is not provided, WIN.INI is used. If the specified destination file does not exist, it is created. If an .INI setting is not present in the destination file, it is added. The function returns TRUE if it completes successfully or FALSE if it doesn't.

It is assumed that all entries in the `srcfile$` .INI file are in standard .INI format. Any entries that do not follow this Windows standard are ignored. If the specified source file does not exist, the function ends.

Example

```
sub main()
   'Example of AddIni

 'add the contents of myini.ini to win.ini
   a% = AddIni("myini.ini","win.ini")
   if a% then
      msgbox "INI settings added to Win.INI"
   else
      msgbox "AddIni failed."
   end if
end sub
```

See Also Environment Statements and Functions (Chapter 7)

And

Description And operator.

Syntax `expression1 And expression2`

Returns TRUE if `expression1` is TRUE and `expression2` is TRUE, otherwise FALSE.

If the two operands are numeric, then the result is the bitwise AND of the two arguments.

Notes If either of the two operands is a floating point number, then the two operands are first converted to longs, then a bitwise AND is performed.

Example

```
sub main()
   'AND statement
   dim a as integer
   dim b as integer

   a = 5
   b = 9
   if (a < 6) AND (b > 8) then
      msgbox "Both conditions were true."
   else
      msgbox "Both  conditions were not true."
   end if

   if (a < 6) AND (b > 9) then
      msgbox "Both conditions were true."
   else
      msgbox "Both conditions were not true."
   end if
end sub
```

See Also Operators (Chapter 7)

AnswerBox Function

Description Displays a box that prompts the user for a response and indicates which button the user pressed.

Syntax `AnswerBox(prompt$ [,button1$ [,button2$ [,button3$]]])`

Returns Returns an integer indicating which button was pushed (1 for the first button, 2 for the second, and so on). 0 is returned if the user cancels the dialog box (by pressing Escape).

Comments The dialog box is sized to hold the entire contents of `prompt$`. The `prompt$` string can contain CR/LF, Chr$(13)+Chr$(10), to separate lines.

The maximum size of the dialog box is 5/8 the width of the screen and 3/4 the height of the screen. If a given line is too long, it will be word wrapped.

The `button$` parameters specify one or more buttons to appear below the displayed `prompt$`. If no buttons are specified, then "OK" and "Cancel" are used. Up to three buttons can be specified. The width of each button is determined by the width of the widest button.

You can use the '&' symbol to specify an accelerator key in the label of the button.

The dialog box uses the 8 point Helvetica font.

Example

```
sub main()
   'AnswerBox example

   'This is a default answer box without
'specifying the buttons
   r% = AnswerBox("Example prompt?")
   msgbox str$(r%)    'display the result
   r% = AnswerBox("Example prompt
2?","&Maybe","&OK","Maybe
Not")
   msgbox str$(r%)   'display the result
end sub
```

See Also Dialog Display (Chapter 7)

AppActivate Statement

Description Activates the specified top-level window.

Syntax `AppActivate WindowName$`

Comments The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If the application is minimized, it will be restored by this command. If the window being activated is a full-screen DOS application, that application will be restored to full screen.

A runtime error results if the window being activated is not enabled (which is the case if that application currently is displaying a modal dialog box) or if the window is not found.

Example

```
sub main()
   'AppActivate example

   'activate Applications Manager
   AppActivate "Applications Manager"
end sub
```

See Also Window Manipulation (Chapter 7)

AppClose Statement

Description Closes the specified top-level application.

Syntax `AppClose [WindowName$]`

Comments The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, the window with the focus is used (i.e., the active window).

A runtime error results if the window being activated is not enabled (which is the case if that application currently is displaying a modal dialog box).

Example

```
sub main()
   'AppClose example

   'search for a copy of Notepad that is
running
   appname$ = AppFind$("Notepad")

   'close the copy of Note pad that was found
   AppClose appname$
end sub
```

See Also Window Manipulation (Chapter 7)

AppFileName$ Function

Description Returns the filename of the program that owns the top-level window of the given title.

Syntax `AppFileName$(WindowName$)`

Comments The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

For DOS applications, the filename returned is that of the actual executable program.

Example

```
sub main()
   'example of AppFileName$

   'find a copy of Notepad that is running
   appname$ = AppFind$("Notepad")

   'get the name of the program it was
executed from
   appfile$ = AppFileName$(appname$)

   'display the file name
   MsgBox appfile$
end sub
```

See Also Window Manipulation (Chapter 7)

AppFind Function

Description Returns the full name of the top-level window, given a partial window name.

Syntax `AppFind$(partial_name$)`

Comments The name is specified using the same format as that used by the `WinActivate` statement.

Example

```
sub main()
   'example of AppFind$
   appname$ = AppFind$("Notepad")
   MsgBox appname$
end sub
```

See Also Window Manipulation (Chapter 7)

AppGetActive$ Function

Description Returns the title of the active window.

Syntax `AppGetActive$()`

Comments This function is used to retrieve the title of the active top-level window. The returned value can be used in subsequent calls to routines that require a title to a window.

If "" is returned, no window is active. This is a rare occurrence, it indicates that Windows may be in an unstable state.

Example 1

```
sub main()
   'example of AppGetActive$

   'activate a copy of Notepad
   appname$ = AppFind$("Notepad")
   AppActivate appname$

   'now see if it is active
   appname$ = AppGetActive$()
   MsgBox appname$
end sub
```

Example 2

```
n$ = AppGetActivate$
AppMinimize n$
```

See Also Window Manipulation (Chapter 7)

AppGetPosition Statement

Description Gets the position of a specified top-level window.

Syntax `AppGetPosition x%,y%,width%,height% [,WindowName$]`

Comments The numeric arguments are filled with the pixel position of the window on the display. If an argument is not a variable reference, then the argument is ignored, as in the following example which only retrieves the position and ignores the width and height:

dim x as integer, y as integer
AppGetPosition x,y,0,0,"Program Manager"

The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, then the window with the focus is used (i.e., the active window).

Example

```
sub main()
   'example of AppGetPosition
   dim x, y, w, h as integer

   'get the position, width and height of
Notepad
   appname$ = AppFind$("Notepad")
   AppGetPosition x, y, w, h, appname$

   'display them
   MsgBox appname$+" is at
"+str$(x)+","+str$(y)+ " with width of "+  ↵
      str$(w)+" and height of "+str$(h)
end sub
```

See Also Window Manipulation (Chapter 7)

AppGetState Function

Description Returns an integer representing the state of the top-level window.

Syntax `AppGetState(WindowName$)`

Returns

`WS_MAXIMIZED`	window is maximized
`WS_MINIMIZED`	window is minimized
`WS_RESTORED`	window is restored

Comments The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, then the window with the focus is used (i.e., the active window).

Example

```
sub main()
   'example of AppGetState

   appname$ = AppFind$("Notepad")
   appstate% = AppGetState(appname$)
   if appstate% = WS_MAXIMIZED then
      MsgBox appname$ + " is Maximized - " +
str$(appstate%)
   else
      if appstate% = WS_MINIMIZED then
         MsgBox appname$ + " is Minimized - "
+ str$(appstate%)
      else
         if appstate% = WS_RESTORED then
            MsgBox appname$ + " is Restored -
" +
tr$(appstate%)
         else
            MsgBox appname$ + " is in an
unknown state - " +
tr$(appstate%)
         end if
      end if
   end if
end sub
```

See Also Window Manipulation (Chapter 7)

AppHide Statement

Description Hides the specified top-level window.

Syntax `AppHide [WindowName$]`

Comments Nothing happens if the window is already hidden.

The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, the window with the focus is used (i.e., the active window).

A runtime error results if the window being activated is not enabled (which is the case if that application currently is displaying a modal dialog box).

Example

```
sub main()
   'example of AppHide and AppShow
   appname$ = AppFind$("Notepad")
   AppHide appname$
   MsgBox appname$+" is now hidden."
   AppShow appname$
   MsgBox appname$+" is no longer hidden."
end sub
```

See Also Window Manipulation (Chapter 7)

AppList Statement

Description Fills the specified string array with the names of all the active applications.

Syntax `AppList ArrayofAppNames$()`

Comments Before calling this function, you must declare the array to contain the names of the application. Use the `Dim` command.

After calling this function, use the `lbound()` and `ubound()` functions to determine the new size of the array.

Example

```
sub main()
   'example of AppList
   dim appnames(1) as string
   AppList appnames
   for i% = lbound(appnames) to
ubound(appnames)
      MsgBox appnames(i%)
   next I%
end sub
```

See Also Window Manipulation (Chapter 7)

AppMaximize Statement

Description Maximizes the specified top-level window.

Syntax `AppMaximize [WindowName$]`

Comments Nothing happens if the window is already maximized or is hidden.

The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, the window with the focus is used (i.e., the active window).

A runtime error results if the window being activated is not enabled (which is the case if that application currently is displaying a modal dialog box).

Example

```
sub main()
   'example of AppMaximize, AppRest ore, and

'AppMinimize

appname$ = AppFind$("Notepad")
   AppMaximize appname$
   msgbox appname$+" is maximized."
   AppRestore appname$
   msgbox appname$+" is restored."
   AppMinimize appname$
   msgbox appname$+" is minimized."
end sub
```

See Also Window Manipulation (Chapter 7)

AppMinimize Statement

Description Minimizes the specified top-level window.

Syntax `AppMinimize [WindowName$]`

Comments Nothing happens if the window is already minimized or is hidden.

The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, the window with the focus is used (i.e., the active window).

A runtime error results if the window being activated is not enabled (which is the case if that application currently is displaying a modal dialog box).

Example

```
sub main()
   'example of AppMaximize, AppRestore, and

'AppMinimize

   appname$ = AppFind$("Notepad")
   AppMaximize appname$
   msgbox appname$+" is maximize d."
   AppRestore appname$
   msgbox appname$+" is restored."
   AppMinimize appname$
   msgbox appname$+" is minimized."
end sub
```

See Also Window Manipulation (Chapter 7)

AppMove Statement

Description Sets the position of the specified top-level window to a given `x,y` pixel location.

Syntax `AppMove x%,y%[,WindowName$]`

Comments This statement has no effect if the top-level window is maximized.

It is valid to specify an `x,y` location that is not visible.

The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, the window with the focus is used (i.e., the active window).

A runtime error results if the window being activated is not enabled (which is the case if that application currently is displaying a modal dialog box).

Example

```
sub main()
   'example of AppMove
   dim ax, ay as integer

   '  get current position of Notepad to save
   appname$ = AppFind$("Notepad")
   AppGetPosition ax, ay, 0, 0,  appname$
```

```
   '  move Notepad around a little
   for i% = 0 to 200
      AppMove i%, i%, appname$
   next I%

   AppMove ax, ay, appname$
end sub
```

See Also Window Manipulation (Chapter 7)

AppRestore Statement

Description Restores the specified top-level window.

Syntax `AppRestore [WindowName$]`

Comments This statement has an effect only if the specified window is either maximized or minimized.

`AppRestore` will do nothing if the specified window is hidden.

The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, the window with the focus is used (i.e., the active window).

A runtime error results if the window being activated is not enabled (which is the case if that application currently is displaying a modal dialog box).

Example

```
sub main()
   'example of AppMaximize, AppRestore, and
   'AppMinimize

   appname$ = AppFind$("Notepad")
   AppMaximize appname$
   msgbox appname$+" is maximized."
   AppRestore appname$
   msgbox appname$+" is restored."
   AppMinimize appname$
   msgbox appname$+" is minimized."
end sub
```

See Also Window Manipulation (Chapter 7)

AppSetState Statement

Description Sets the state of the specified top-level window.

Syntax `AppSetState [WindowName$]`

Comments This statement sets the state of the specified top-level window to one of the following values:

`WS_MAXIMIZED`	maximize the window
`WS_MINIMIZED`	minimize the window
`WS_RESTORED`	restore the window

The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, the window with the focus is used (i.e., the active window).

Example

```
sub main()
   'example of AppSetState

   appname$ = AppFind$("Notepad")
   AppSetState WS_MAXIMIZED, appname$
   msgbox "Notepad is maximized"
   AppSetState WS_RESTORED, appname$
   msgbox "Notepad is restored"
   AppSetState WS_MINIMIZED, appname$
   msgbox "Notepad is minimized"
end sub
```

See Also Window Manipulation (Chapter 7)

AppShow Statement

Description Shows the specified top-level window.

Syntax `AppShow [WindowName$]`

Comments Nothing happens if the window is already displayed.

The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, the window with the focus is used (i.e., the active window).

A runtime error results if the window being activated is not enabled (which is the case if that application currently is displaying a modal dialog box).

Example

```
sub main()
   'example of AppHide and AppShow

   appname$ = AppFind$("Notepad")
   AppHide appname$
   MsgBox appname$+" is now hidden."
   AppShow appname$
   MsgBox appname$+" is no longer hidden."
end sub
```

See Also Window Manipulation (Chapter 7)

AppSize Statement

Description Sets the width and height of the specified top-level window. It lets you size sizable and non-sizable windows.

Syntax `AppSize width%,height%[,WindowName$]`

Comments This statement works only if the specified application is restored (i.e., not minimized or maximized).

The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, the window with the focus is used (i.e., the active window).

A runtime error results if the window being activated is not enabled (which is the case if that application currently is displaying a modal dialog box).

Example

```
sub main()
   'example of AppSize
   dim ax, ay, aw, ah as integer

   appname$ = AppFind$("Notepad")
   AppSetState WS_RESTORED, appname$
   AppGetPosition ax, ay, aw, ah, appname$
   for i% = 10 to 200
      AppSize i%, i%, appname$
   next I%
   AppSize aw, ah, appname$
   AppMove ax, ay, appname$
   AppSetState WS_MINIMIZED, appname$
end sub
```

See Also Window Manipulation (Chapter 7)

AppType Function

Description Returns an integer representing the type of application owning the specified window.

Syntax `AppType[(WindowName$)]`

Returns

`TYPE_DOS`	DOS executable
`TYPE_WINDOWS`	Windows executable

Comments The `WindowName$` parameter is the title of a top-level window. The complete text of the window title is required. The title is not case-sensitive.

If `WindowName$` parameter is missing, the window with the focus is used (i.e., the active window).

Example

```
sub main()
   'example of AppType

   msgbox "After you press OK, you have 10
seconds to activate
n app to ↵
      check."
   sleep 10000    'wait for 10 seconds
   appname$ = AppGetActive$()
   at% = AppType(appname$)
```

```
      select case at%
         case TYPE_DOS
            msgbox "DOS Application"
         case TYPE_WINDOWS
            msgbox "WINDOWS Application"
      end select
   end sub
```

See Also Window Manipulation (Chapter 7)

ArrayDims Function

Description Returns an integer representing the number of dimensions in the specified array.

Syntax `ArrayDims(`*`arrayvariable`*`)`

Comments If the function indicates zero dimensions, the array was declared as an empty array (e.g. `dim x$()`).

Example 1

```
sub main()
   'example of ArrayDims
   dim apps(1) as string
   dim oapps(1,1) as string

   'ArrayDims shows how many dimensions an
array
'was declared with
   msgbox "apps(1) has
"+str$(arraydims(apps))+"
imension(s)."
   msgbox "oapps(1,1) has
+str$(arraydims(oapps))+"dimension(s)."
end sub
```

Example 2

```
sub main()
dim f$()       'allocate empty array
   files f$,"C:\*.BAT"   'fill the array
   if dims(f$) = 0 then exit sub  'exit if no
                        'elements
end sub
```

See Also Arrays (Chapter 7)

ArraySort Statement

Description Sorts a single-dimensioned array.

Syntax 1 `ArraySort s$()`

Syntax 2 `ArraySort a%()`

Syntax 3 `ArraySort a&()`

Syntax 4 `ArraySort a!()`

Syntax 5 `ArraySort a#()`

Comments If a string array is specified, then the routine sorts alphabetically in ascending order (using case-sensitive string comparisons). If a numeric array is specified, the routine sorts lower numbers to the lowest array index locations.

A runtime error results if an array with more than one dimension is specified.

Example 1

```
sub main()
   'example of ArraySort
   dim apps(1) as string

   'get list of running applications
   AppList apps
   'sort it
   ArraySort apps
   'display them
   for i% = lbound(apps) to ubound(apps)
      msgbox apps(i%)
   next i%
end sub
```

Example 2

```
sub main()
   dim a$(100)
   ArraySort a$
   result = SelectBox("Title","Prompt:",a$)
end sub
```

See Also Arrays (Chapter 7)

Asc Function

Description Returns the numeric ASCII code for the first character of the specified string.

Syntax `asc(text$)`

Comments The return value is an integer between 0 and 255.

Example

```
sub main()
   'example of ASC()
   dim acode as integer
   dim astr as string
   astr = "This is a string."
   'the ASC function returns the ASCII code
for
   'the first character of the given string
   acode = asc(astr)
   msgbox "The first character of the string
("+astr+") has an ASCII code  ↵
      of "+str$(acode)
end sub
```

See Also Conversions (Chapter 7)

Strings (Chapter 7)

AskBox$ Function

Description Displays a dialog box that asks the user to enter a string in an edit box, and returns the string the user enters.

Syntax `AskBox$(prompt$ [,default$])`

Returns Returns a string that the user typed, or returns an empty string indicating that the user canceled the dialog box.

Comments The dialog box is sized to the appropriate width depending on the width of `prompt`.

When the dialog box is displayed, the edit box has the focus.

If `default` is specified, then this is used as the initial contents of the edit field.

The dialog box has OK and Cancel buttons.

The dialog box uses the 8 point Helvetica font.

The maximum number of characters that can be typed into the edit box is 255.

Example

```
sub main()
   'example of AskBox$

   userinput$ = AskBox$("Key in something
below:","This is the default  ↵
      value.")
   msgbox "You entered -> "+userinput$
end sub
```

See Also Dialog Display (Chapter 7)

AskPassword$ Function

Description Displays a dialog box that asks the user to enter his or her password in an edit box, and returns the string the user enters.

Syntax `AskPassword$(prompt$)`

Returns Returns the string that the user typed, or returns an empty string indicating that the user canceled the dialog box.

Comments Unlike the `AskBox` command, the user sees asterisks in place of the characters that are actually typed. This allows the input of passwords.

When the dialog box is displayed, the edit box has the focus.

The dialog box is sized to the appropriate width depending on the width of `prompt$`.

The dialog box has OK and Cancel buttons.

The dialog box uses the 8 point Helvetica font.

The maximum number of characters that can be typed into the edit box is 255.

Example

```
sub main()
   'example of AskPassword$

   mypassword$ = AskPassword$("Type in a fake
password:")
   msgbox "The password you typed was ->
"+mypassword$
end sub
```

See Also Dialog Display (Chapter 7)

Atn Function

Description Returns a double-precision number representing the arctangent of a given number.

Syntax `atn(number#)`

Comments To calculate the value of PI, use:

```
4 * ATN(1)
```

Example

```
sub main()
   'example of ATN (ar ctangent)
   dim mypi as double

   mypi = 4 * atn(1)
   msgbox str$(mypi)
end sub
```

See Also Math Statements and Functions (Chapter 7)

ATTR_ARCHIVE

Description Constant.

Value 32

Comments Bit position of a file attribute indicating that a file hasn't been backed up. This value is used by `GetAttr`, `SetAttr`, and `FileList`.

ATTR_DIRECTORY

Description Constant.

Value 16

Comments Bit position of a file attribute indicating that a file is a directory entry. This value is used by `GetAttr`, `SetAttr`, and `FileList`.

ATTR_HIDDEN

Description Constant.

Value 2

Comments Bit position of a file attribute indicating that a file is hidden. This value is used by `GetAttr`, `SetAttr`, and `FileList`.

ATTR_NONE

Description Constant.

Value 64

Comments Bit position of a file attribute indicating that a file has no other attributes set. This value is used by `GetAttr`, `SetAttr`, and `FileList`.

ATTR_NORMAL

Description Constant.

Value `0`

Comments Bit position of a file attribute indicating that a file has no other attributes set. This value is used by `GetAttr`, `SetAttr`, and `FileList`.

ATTR_READONLY

Description Constant.

Value `1`

Comments Bit position of a file attribute indicating that a file is read-only. This value is used by `GetAttr`, `SetAttr`, and `FileList`.

ATTR_SYSTEM

Description Constant.

Value `4`

Comments Bit position of a file attribute indicating that a file is a system file. This value is used by `GetAttr`, `SetAttr`, and `FileList`.

ATTR_VOLUME

Description Constant.

Value `8`

Comments Bit position of a file attribute indicating that a file is the volume label. This value is used by `GetAttr`, `SetAttr`, and `FileList`.

Beep Statement

Description Makes a single system beep.

Syntax `beep`

Example

```
sub main()
   'example of BEEP

   beep
end sub
```

Begin Dialog...End Dialog Statement

Description The `Begin Dialog...End Dialog` block defines a dialog box template.

Syntax

```
begin dialog DialogName$,x%,y%,width%,height%
        ...
        ...
end dialog
```

Comments The `x,y` parameters are the dialog coordinates of the upper left hand corner of the dialog box relative to the parent window.

The `width,height` parameters specify the width and height dimensions of the dialog box in dialog units.

The `DialogName` parameter specifies the name used to dimension a variable of this type (i.e., a variable that refers to this dialog box template). Once a dialog template has been defined, a variable can be dimensioned using this name:

```
dim MyDlg as DialogName
```

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

A dialog template must have at least one `PushButton`, `OKButton`, or `CancelButton`. Otherwise, there will be no way to close the dialog box.

Dialog units are defined as 1/4 the width of the font in the horizontal direction and 1/8 the height of the font in the vertical direction. All dialogs created by DCL use an 8 point Helvetica font.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

ButtonEnabled Function

Description Determines whether the specified button within the current window is enabled.

Syntax 1 `ButtonEnabled(ButtonName$)`

Syntax 2 `ButtonEnabled(ButtonID%)`

Returns Returns the integer TRUE if the specified button within the current window is enabled, otherwise this function returns FALSE.

Comments The button can be specified either by its name (`ButtonName$`) or using its ID (`ButtonID%`). `ButtonName$` is the text on the button's label.

When a button is enabled, it can be pressed using the `SelectButton` statement.

Example

```
sub main()
   'Example of ButtonEnabled

   appn$ = AppFind$("Notepad")
   if appn$ <> "" then
      AppActivate appn$
      Menu "File.Page Setup"
      if ButtonEnabled("OK") then
         msgbox "OK button is enabled."
      else
         msgbox "OK button is not enabled."
      end if
   end if
end sub
```

See Also Dialog Manipulation (Chapter 7)

ButtonExists Function

Description Determines whether the specified button exists within the current window.

Syntax 1 `ButtonExists(ButtonName$)`

Syntax 2 `ButtonExists(ButtonID%)`

Returns Returns the integer TRUE if the specified button exists within the current window, otherwise this function returns FALSE.

Comments The button can be specified either by its name (`ButtonName$`) or using its id (`ButtonID%`). `ButtonName$` is the text on the button's label.

Example

```
sub main()
   'Example of ButtonExists

   appn$ = AppFind$("Notepad")
   if appn$ <> "" then
      AppActivate appn$
      Menu "File.Page Setup"
      if ButtonExists("Cancel") then
         msgbox "Cancel button exists."
      else
         msgbox "Cancel button does not
exist."
      end if
   end if
end sub
```

See Also Dialog Manipulation (Chapter 7)

Call Statement

Description Transfers control to the specified subroutine, optionally passing arguments.

Syntax `call subroutine_name [(arguments)]`

Comments Using this statement is equivalent to:

`subroutine_name [arguments]`

Use of the `call` statement is never required.

Example

```
sub doubleit (a as integer)
   msgbox str$(a * 2)
end sub
sub main()
   'Example of the Call statement

   'the following statements do the same
thing
   Call doubleit (3)
   doubleit(3)
end sub
```

See Also Procedure Statements (Chapter 7)

CancelButton Statement

Description Defines a Cancel button that appears within a dialog box template.

Syntax `CancelButton x%,y%,width%,height%`

Comments This statement can only appear within a dialog box template definition (BEGIN DIALOG...END DIALOG).

The `x,y,width,height` parameters are specified in dialog coordinates. The `x,y` position is relative to the upper left corner of the dialog box.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

CDbl Function

Description Returns the double-precision equivalent of the passed numeric expression.

Syntax `cdbl(number#)`

Comments This function has the same effect as assigning the numeric expression to a double-precision variable.

Example

```
sub main()
   'example of CDBL
   dim adouble as double
   dim asingle as single
   dim ainteger as integer
   dim along as long asingle = pi
   adouble = pi
   ainteger = 30000
   along = 88000
   adouble = cdbl(ainteger)
   msgbox str$(adouble)
```

```
   adouble = cdbl(along)
   msgbox str$(adouble)
   msgbox str$(asingle)
   adouble = pi
   msgbox str$(adouble)
   'Now convert the single to double. You'll
see a
   'change in the value due to conversion of
a
   'smaller precision to double-precision.
   adouble = cdbl(asingle)
   msgbox str$(adouble)
end sub
```

See Also Conversions (Chapter 7)

ChDir Statement

Description Changes the current directory of the specified drive.

Syntax `chdir newdir$`

Comments This statement will not change to the specified drive.

If you do not include a drive in the `newdir$` parameter, the current drive is assumed.

This statement behaves the same as the DOS "cd" command.

Example

```
sub main()
   'example of ChDir

   msgbox CurDir$("C")
   ChDir "C:\"
   msgbox CurDir$("C")
end sub
```

See Also File Input and Output (Chapter 7)

ChDrive Statement

Description Changes the default drive.

Syntax `chdrive DriveLetter$`

Comments Only the first character of `DriveLetter$` is used. You can use the same string for the `ChDrive` and `ChDir` commands.

`DriveLetter$` is case insensitive.

If `DriveLetter$` is empty, then the current drive is not changed.

Example

```
sub main()
   'example of ChDrive

   msgbox CurDir$()
   chdrive "D"
   msgbox CurDir$()
end sub
```

See Also File Input and Output (Chapter 7)

CheckBox Statement

Description Defines a checkbox within a dialog box template.

Syntax `CheckBox x%,y%,width%,height%,title$,.Field`

Comments Checkboxes can be either on or off, depending on the value of `.Field`.

This statement can only appear within a dialog box template definition (BEGIN DIALOG...END DIALOG).

The `x,y,width,height` parameters are specified in dialog coordinates. The `x,y` position is relative to the upper left corner of the dialog box. The `width` and `height` parameters specify the dimensions of the checkbox and label.

On entry to the `Dialog` statement, the `.Field` variable is used to set the initial state of the checkbox. On exit from the `Dialog` statement, the `.Field` variable is used to determine the final state of the checkbox. If the value is 0, the checkbox is unchecked; 1 indicates that the checkbox is checked.

The `title$` parameter may contain an ampersand character to denote an underlined accelerator, such as "&Font" for `Font`.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

CheckboxEnabled Function

Description Determines whether the specified checkbox in the current window is enabled.

Syntax 1 `CheckboxEnabled(CheckboxName$)`

Syntax 2 `CheckboxEnabled(CheckboxID%)`

Returns Returns the integer TRUE if the specified checkbox within the current window is enabled, otherwise this function returns FALSE.

Comments The checkbox can be specified either by its name (`CheckboxName$`) or using its id (`CheckboxID%`). `CheckboxName$` is the text of the checkbox label.

When a checkbox is enabled, its state can be set using the `SetCheckbox` statement.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

CheckboxExists Function

Description Determines whether the specified checkbox exists within the current window.

Syntax 1 `CheckboxExists(CheckboxName$)`

Syntax 2 `CheckboxExists(CheckboxID%)`

Returns Returns the integer TRUE if the specified checkbox exists within the current window, otherwise this function returns FALSE.

Comments The checkbox can be specified either by its name (`CheckboxName$`) or using its ID (`CheckboxID%`). `CheckboxName$` is the text of the checkbox label.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

Chr$ Function

Description Returns the character for the specified ASCII code.

Syntax `chr$(AsciiCode%)`

Comments `AsciiCode` must be an integer between 0 and 255.

The `Chr$()` function can be used within constant declarations, as in the following example:

```
const crlf$ = chr$(13) + chr$(10)
```

Example

```
sub main()
   'example of Chr$()

   'List the letters of the alphabet
   startcode = asc("A")
   alphstr$ = ""
   msgbox "The alphabet starts at code
"+str$(startcode)
   for i% = startcode to startcode + 25
      alphstr$ = alphstr$ + chr$(i%)
   next i%
   msgbox alphstr$
end sub
```

See Also Conversions and Strings (Chapter 7)

CInt Function

Description Returns the integer portion of the given expression. In cases of fractions, this function rounds to the nearest integer.

Syntax `cint(number#)`

Comments The passed numeric expression must be within the following range:

```
-32768 <= number <= 32767
```

A runtime error results if the passed expression is not within the above range.

This function has the same effect as assigning a numeric expression to a variable of type integer.

Example

```
sub main()
   'Example of CINT
   dim a as integer

   a = cint(pi)   'convert PI to an integer
   msgbox str$(a)
end sub
```

See Also Conversions (Chapter 7)
Fix Function
Int Function

Clipboard$ Statement and Function

Description The `Clipboard$` statement puts a string in the clipboard.

The `Clipboard$` function returns the text in the clipboard.

Statement Syntax
```
clipboard$ NewContent$
```

Comments This statement puts `NewContent$` into the clipboard.

Function Syntax
```
clipboard$()
```

Returns Text contained in the clipboard.

Comments If the clipboard doesn't contain text, or the clipboard is empty, then an empty string is returned.

Example
```
sub main()
   'Example of Clipboard$(), Clipboard$, and
   'ClipboardClear
   'WARNING: after you run this, your
clipboard
   'contents will be gone

   msgbox Clipboard$()
   ClipboardClear
   msgbox Clipboard$()
   Clipboard$ "I was here!"
   msgbox Clipboard$()
   ClipboardClear
end sub
```

See Also Clipboard Manipulation (Chapter 7)

ClipboardClear Statement

Description Clears (removes the contents of) the clipboard.

Syntax
```
ClipboardClear
```

Example

```
sub main()
   'Example of Clipboard$(), Clipboard$, and
   'ClipboardClear
   'WARNING: after you run this, your
clipboard
   'contents will be gone

   msgbox Clipboard$()
   ClipboardClear
   msgbox Clipboard$()
   Clipboard$ "I was here!"
   msgbox Clipboard$()
   ClipboardClear
end sub
```

See Also Clipboard Manipulation (Chapter 7)

CLng Function

Description Returns a long integer representing the result of the given numeric expression.

Syntax `Clng(number#)`

Comments The passed numeric expression must be within the following range:

```
-2147483648 <= number <= 2147483647
```

A runtime error results if the passed expression is not within the above range.

This function has the same effect as assigning a numeric expression to a long variable.

Example

```
sub main()
   'Example of CLNG
   dim a as long

   a = clng(pi)   'convert PI to a long
integer
   msgbox str$(a)
end sub
```

See Also Conversions (Chapter 7)

Close Statement

Description Closes open file(s).

Syntax `Close [[#] filenumber% [,[#] filenumber%]]`

Comments The file number is assigned by the `Open` command.

If no arguments are specified, this statement closes all files. Otherwise, this statement closes each specified file.

Example Input/Output Example

See Also File Input and Output (Chapter 7)

Combobox Statement

Description Defines a combobox that appears within a dialog box template.

Syntax `ComboBox x%,y%,width%,height%,items$(),.Field`

Comments The `items$` array must be a single-dimension array of strings. The elements of this array are placed into the combobox when the dialog box is created. The `.Field` parameter defines the name used to extract which string occupies the combobox when the dialog box ends. On exit from the `Dialog` statement, the `.Field` contains an index to the item that is highlighted in the combobox.

This statement can only appear within a dialog box template definition (BEGIN DIALOG...END DIALOG).

The `x,y,width,height` parameters are specified in dialog coordinates. The `x,y` position is relative to the upper left corner of the dialog box. The `width` and `height` parameters define the dimensions of the drop-down list box.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

ComboboxEnabled Function

Description Determines whether the specified combobox is enabled in the current window or dialog box.

Syntax 1 `ComboboxEnabled(name$)`

Syntax 2 `ComboboxEnabled(id%)`

Comment The combobox can be specified either by `name$` or `id%`. The `name$` parameter specifies the text that appears in the static control that immediately precedes the combobox control in the window list (or dialog template).

This function returns the integer TRUE if the given combobox is enabled within the current window or dialog box, FALSE otherwise.

A runtime error is generated if the specified combobox does not exist.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

ComboboxExists Function

Description Determines whether the specified combobox exists in the current window or dialog box.

Syntax 1 `ComboboxExists(name$)`

Syntax 2 `ComboboxExists(id%)`

Comments The combobox can be specified either by `name$` or `id%`. The `name$` parameter specifies the text that appears in the static control that immediately precedes the combobox control in the window list (or dialog template).

This function returns the integer TRUE if the given combobox is enabled within the current window or dialog box, FALSE otherwise.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

Command$ Statement

Description Returns a string representing the arguments from the command line used to start the application.

Syntax `Command$`

Comments When running scripts from the DCL editor, you enter command-line arguments through the Arguments command on the Run menu of the DCL editor.

Const Statement

Description Declares a constant for use within the current script.

Syntax `const name = expression [,name = expression]...`

Comments The `name` is only valid within the current DCL script. The `expression` must be assembled from literals, or other constants. Calls to functions are not allowed.

See Also Variables and Constants (Chapter 7)

Cos Function

Description Returns a double-precision number representing the cosine of a given angle.

Syntax `cos(angle#)`

Comments The `angle#` parameter is given in radians.

See Also Math Statements and Functions (Chapter 7)

CSng Function

Description Returns a single-precision number representing the result of the given numeric expression.

Syntax `CSng(number#)`

Comments This function has the same effect as assigning a numeric expression to a single-precision variable.

Example

```
sub main()
   'Example of CSNG()

   dim adouble as double
   dim asingle as single

   adouble =pi
   msgbox str$(adouble)
   asingle = csng(adouble)
   msgbox str$(asingle)
end sub
```

See Also Conversions (Chapter 7)

CStr Function

Description Returns a string representing the result of the given expression.

Syntax `CStr(number#)`

Example

```
sub main()
   'Example of CSTR

   dim adouble as double
   dim astring as string

   adouble = pi
   astring = cstr(adouble)
   msgbox astring
end sub
```

See Also Conversions (Chapter 7)

CurDir$ Function

Description Gets the current directory.

Syntax `curdir$[(drive$)]`

Returns Returns the current directory on the specified drive. If no drive is specified, the current directory on the current drive is returned.

Comments A runtime error results if `drive$` is invalid.

Example

```
sub main()
   'Example of CurDir$()
   dim dstr as string

   dstr = CurDir$()
   msgbox dstr
   dstr = CurDir$("F")   'for a particular
drive
   msgbox dstr
end sub
```

See Also File Input and Output (Chapter 7)

Date$ Statement and Function

Description The `date$` assignment statement sets the system date. The `date$` function returns the system date.

Assignment Syntax

```
date$ = newdate$
```

Comments

```
Sets the system date to the specified date. The format for
newdate$ is any of the following:
MM-DD-YYYY
MM-DD-YY
MM/DD/YYYY
MM/DD/YY
```

Example 1

```
sub main()
   'Example of the Date$ command (not
function)
   dim currentdate as string
   dim newdate as string
   currentdate = Date$
   newdate = "1-1-1980"    'This is the
earliest
                           'date that can be
                           'assigned
   Date$ = newdate
   msgbox Date$
   Date$ = currentdate
   msgbox Date$
end sub
```

Example 2

```
date$ = "7-28-1992"
```

Function Syntax

```
Date$[()]
```

Returns The `date$` function returns the current system date as a 10 character string.

Comments The format for the returned date is `MM-DD-YYYY`.

Example

```
sub main()
   'Example of Date$()
   dim dstr as string

   dstr = Date$
   msgbox dstr
   dstr = Date$()
   msgbox dstr
end sub
```

See Also Date and Time Functions (Chapter 7)

DateSerial Function

Description Returns a double-precision number representing the specified date. The number is returned in days where Dec 30, 1899 is 0.

Syntax `DateSerial(year%,month%,day%)`

Example

```
sub main()
   'example of DateSeria l
   dim dserial as double

   dserial = DateSerial(1899,12,30) 'Earliest
date should be 0
   msgbox str$(dserial)
   dserial = DateSerial(1999,12,30)  '100
years later (leap years included)
   msgbox str$(dserial)
end sub
```

See Also Conversions (Chapter 7)

Date and Time Functions (Chapter 7)

DateValue Function

Description Returns a double-precision number representing the date contained in the specified string argument.

Syntax `DateValue(date_string$)`

Comments This function interprets the passed `date_string$` parameter looking for a valid date specification. Date specifications vary depending on the international settings contained in the INTL section of the WIN.INI file.

The `date_string$` parameter can contain valid date items separated by date separators such as slash (/), minus (-), or comma (,). The date items must follow the ordering determined by the current date format settings in use by Windows.

Date strings can contain an optional time specification, but this is not used in the formation of the returned value.

Months can appear in their abbreviated formats, such as Jan, Feb, or Mar.

Example

```
sub main()
   'Example of DateValue
   dim dval as double

   dval = DateValue(Date$)  'value of current
date
   msgbox str$(dval)
end sub
```

See Also Conversions (Chapter 7)

Date and Time Functions (Chapter 7)

Day Function

Description Returns an integer representing the day of the month for the date encoded in the specified `serial` parameter. The value returned is between 1 and 31 inclusive.

Syntax `Day(serial#)`

Comments You can obtain the value for the `serial#` parameter by using the `DateSerial` or `DateValue` command.

Example

```
sub main()
   'Example of Day() function (day of month)
   'will show the current date's day of
month.
   msgbox str$(Day(DateValue(Date$)))
end sub
```

See Also Date and Time Functions (Chapter 7)

DCLHomeDir$ Function

Description Returns the directory containing DCL.

Syntax `DCLHomeDir$()`

Comments This function is used to locate files that are part of the DCL system itself.

Example

```
sub main()
   'DCLHomeDir$ example

   a$ = DCLHomeDir$
   msgbox a$
end sub
```

See Also DCL Environment Information (Chapter 7)

DCLOS$ Function

Description Returns a number indicating the host operating environment.

Syntax `DCLOS$()`

Returns

`0`	Windows
`1`	DOS

Example

```
sub main()
   'Example of DCLOS() function

   a% = DCLOS()
   select case a%
      case 0
         msgbox "Windows"
      case 1
         msgbox "DOS"
   end select
end sub
```

See Also DCL Environment Information (Chapter 7)

DCLVersion$ Function

Description Returns the version of DCL.

Syntax `DCLVersion$()`

Comments This function returns the major and minor version numbers in the format `major.minor`, as in "1.1".

Example

```
sub main()
   'Example of DCLVersion$()

   a$ = DCLVersion$()
   msgbox a$
end sub
```

See Also DCL Environment Information (Chapter 7)

DDEExecute Statement

Description Sends an execute message to another application.

Syntax `DDEExecute channel%,command$`

Comments The `channel` must first be initiated using DDEInitiate. An error will result if `channel` is invalid.

If the receiving application does not execute the instructions, a runtime error will be generated.

The format of `command$` depends on the receiving application.

Example DDE Example

See Also Dynamic Data Exchange (Chapter 7)

DDEInitiate Function

Description Initializes a DDE link to another application.

Syntax `DDEInitiate(app$,topic$)`

Returns Returns a unique number used to subsequently refer to the open DDE channel.

Comments The function returns 0 if the link cannot be established. This will occur under the following circumstances:

- The specified application is not running
- The topic was invalid for that application
- Insufficient memory or system resources to establish DDE link

Example DDE Example

See Also Dynamic Data Exchange (Chapter 7)

DDEPoke Statement

Description Sets the value of a data item in the receiving application associated with an open DDE link.

Syntax `DDEPoke channel%,dataItem$,value$`

Comments The `channel` must first be initiated using DDEInitiate. An error will result if `channel` is invalid.

The format for `dateItem$` and `value$` depends on the receiving application.

Example DDE Example

See Also Dynamic Data Exchange (Chapter 7)

DDERequest Function

Description Gets a data item from a receiving application.

Syntax `DDERequest$(channel%,dataItem$)`

Returns Returns a string representing the value of the given data item in the receiving application associated with the open DDE channel.

Comments The `channel` must first be initiated using `DDEInitiate`. An error will result if `channel` is invalid. The formats for `dataItem$` and the returned value depend on the receiving application.

Example DDE Example

See Also Dynamic Data Exchange (Chapter 7)

DDETerminate Statement

Description Closes the specified DDE channel.

Syntax `DDETerminate channel%`

Comments The `channel` must first be initiated using `DDEInitiate`. An error will result if `channel` is invalid. All open DDE channels are automatically terminated when the script ends.

Example DDE Example

See Also Dynamic Data Exchange (Chapter 7)

DDETerminateAll Statement

Description Closes all currently open DDE channels.

Syntax `DDETerminateAll`

Comments If you do not issue this statement, all open DDE channels are automatically terminated when the script ends.

Example DDE Example

See Also Dynamic Data Exchange (Chapter 7)

DDETimeOut Statement

Description Sets the number of milliseconds that must elapse before a DDE command times out.

Syntax `DDETimeOut milliseconds&`

Comments The default is 10000 (10 seconds).

The timeout should be set before issuing other DDE commands.

Example DDE Example

See Also Dynamic Data Exchange (Chapter 7)

DDE Example

```
sub main()
   '  This script illustrates the use of the DDE commands.
   '  The current shell is used to demonstrate this.
   '  NetTools Applications Manager is assumed to be the
   '  shell.
   '
   '  Demonstrated in this script are:
   '  DDEInitiate, DDETimeout, DDERequest, DDETerminate,
   '  DDEExecute, and DDETerminateAll

   dim channel as integer
   dim winshell as string
   dim topic as string
   dim reqstr as string
   dim grouplist() a s string
   dim i as integer
   dim nl as integer
   dim selitem as integer
   dim groupname as string

   winshell = "AppMan"
   topic = "AppMan"

   '  initiate a DDE conversation with the shell
   channel = DDEInitiate(winshell,topic)
   DDETimeout 5000          'set time out to 5 seconds
                            '(default is 10000
milliseconds)
   if channel = 0 then
      msgbox "Unable to initiate DDE link with
"+winshell+". Script will end now."
      end
   end if
   '  Get a list of the groups in the shell
   reqstr = DDERequest$(channel,"G roups")
   '  Now the group names have to be placed into an array
   '  for use in a Selection Box
   nl = LineCount(reqstr) - 1
   redim grouplist(1 to nl)
   for i = 1 to nl
      grouplist(i) = Line$(reqstr,i)
   next i

   '  Now that we have the groups in a usable list,
   '  allow the user to make selections to
```

```
   '  view the items in the group
viewgroups:
   selitem = SelectBox("Groups",winshell+"
Groups:",grouplist)
   if selitem = 0 then goto shutdown

   '  Get the group name, and enumerate the item names
   groupname = grouplist( selitem)
   reqstr = DDERequest$(channel,groupname)
   '  Display the items in a message box
   msgbox reqstr
   goto viewgroups

shutdown:
   DDETerminate channel
   msgbox "Now we'll create a group, and then delete it."
   channel = DDEInitiate(winshell,topic)
   DDEExecute channel,"[CreateGroup(TestGrp)]"
   DDEExecute channel,"[ShowGroup(TestGrp,1)]"
   msgbox "Look at your shell now and see the new group
before we delete it."
   DDEExecute channel,"[DeleteGroup(TestGrp)]"
   DDETerminateAll
   end
end sub
```

Declare Statement

Description Declares a subroutine or function within another DLL.

Syntax 1
```
Declare sub name [lib libname$ [alias aliasname$]] ↵
 [([argumentlist])]
```

Syntax 2
```
Declare function name [lib libname [alias aliasname$]] ↵
 [([argumentlist])] [as type]
```

Comments This statement must precede any call to an external DLL routine.

The `name` parameter is any DCL valid global name. When declaring functions, a type-declaration character can be included to indicate the return type.

The `libname$` needs to be specified along with an optional `aliasname$`. The `libname$` parameter specifies the DLL that contains the external routine. All of the Windows API routines are contained in DLLs, such as "user", "kernel", "gdi", and so on. The file extension ".EXE" is implied if another extension is not given. If the `libname$` parameter is not the real name of the external routine as it appears within the external DLL, then an alias name must be given providing this name.

For example, the following two statements declare the same routine:

```
declare function GetCurrentTime lib "user"()  as integer

declare function GetTime lib "user" alias  "GetCurrentTime" ↵
        as integer
```

Use an alias when the name of an external routine conflicts with the name of an internal routine or if the external routine name contains invalid characters.

The optional `argumentlist` specifies the arguments received by the routine. The argument list must match exactly with that of the referenced routine. Otherwise unpredictable results may occur. By default, DCL passes arguments by reference. Many DLL routines require a value rather than a reference. The `byval` keyword does this. For example, the following C routine:

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

would be declared as follows:

```
declare sub MessageBeep lib "user" (byval n as integer)
```

The following shows how a C routine that requires a pointer to an integer would be declared (notice the missing `byval` keyword on the third parameter):

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

declare function SystemParametersInfo lib "user" (byval  ↵
        action as integer, byval uParam as integer, pi  ↵
        as integer, byval updateINI as integer)
```

Strings are always passed to DLL routines by reference - the `byval` keyword in this case is unnecessary. If a DLL routine modifies a passed string variable, there must be sufficient space within the string to hold the returned characters. This can be accomplished using the `space$()` function:

```
declare sub GetWindowsDirectory lib "user" (dir$,len%)
            :
            :
        dim s as string
        s = space$(128)
        GetWindowsDirectory s,128
```

For function declarations, the return type can be specified using a type declaration character (i.e., $, %, or &), or by specifying the `[as` *`type`*`]` clause. The valid types are `integer`, `long`, and `string`.

The libraries containing the routines are loaded when the routine is called for the first time (i.e., not when the script is loaded). This allows a script to references external DLLs that potentially do not exist.

The `declare` statement must appear before the function is used.

The `declare` statement is only valid during the life of that script.

Example

```
declare sub MessageBeep lib "user" (byval n
as integer)
declare function GetDebugState lib "user"
alias "GetSystemDebugState" ()  ↵
   as long

sub main()
   'Example of Declare statement
   'All declared external functions and
procedures
   'exist outside of any DCL function or
   'subroutines.
   dim debugstate as long

   MessageBeep(-1)    'standard system beep
   debugstate = GetDebugState()
   msgbox str$(debugstate)
end sub
```

See Also Procedure Statements (Chapter 7)

DEFtype Statement

Description This statement controls automatic type declaration of variables.

Syntax

```
DEFInt letterrange
DEFLng letterrange
DEFStr letterrange
DEFsng letterrange
DEFdbl letterrange
```

Comments Normally, if a variable is encountered that hasn't yet been declared with the `dim` statement, or does not appear with an explicit type declaration character, the variable is declared implicitly as an integer (`DEFInt A-Z`). This can be changed using the `DEF`*`type`* statement to specify starting letter ranges for *`type`* other than integer. The *`letterrange`* parameter is used to specify starting letters. Thus, any variable that begins with a specified character will be declared using the specified *`type`*.

The syntax for *letterrange* is:

```
letter [-letter] [,letter [-letter]]...
```

`DEFtype` variable types are superseded by an explicit type declaration—using a type declaration character or using the `dim` statement.

This statement only affects compiling of scripts.

Example

```
DEFInt i - k, x - z
DEFStr s - v
DEFDbl a - c

sub main()
   'example of DEFtype statement
   'This statement is used to define certain
   'letters of the alphabetic character set
to
   'be used as a certain type of variable.

   i = 1
   msgbox str$(i)
   j = 2.5
   msgbox str$(j)
   s = "test"
   msgbox s
   a = 3
   msgbox str$(a)
   b = 5.6
   msgbox str$(b)
end sub
```

See Also Variables and Constants (Chapter 7)

DesktopCascade Statement

Description Cascades all non-minimized top-level windows.

Syntax `DesktopCascade`

Example

```
sub main()
   'Example of DesktopCascade

   DesktopCascade
end sub
```

See Also Desktop Modifications (Chapter 7)

DesktopSetColors Statement

Description Changes the system colors to one of the predefined color sets in the CONTROL.INI file.

Syntax `DesktopSetColors ControlPanelItemName$`

Example

```
sub main()
   'example of DesktopSetColors

   msgbox "Click to change to WingTips"
   DesktopSetColors "WingTips"
   msgbox "Click to change to Arizona"
   DesktopSetColors "Arizona"
end sub
```

See Also Desktop Modifications (Chapter 7)

DesktopSetWallpaper Statement

Description Changes the Windows wallpaper.

Syntax `DesktopSetWallPaper filename$,tile%`

Comments This statement changes the Windows wallpaper to the bitmap specified by `filename$`. The wallpaper will be tiled is `tile` is TRUE, otherwise the bitmap will be centered on the desktop.

To remove the wallpaper, set the `filename$` parameter to `""`:
`DesktopSetWallPaper "",true`

This statement makes permanent changes to the wallpaper by writing the new wallpaper information to the WIN.INI file.

Example

```
sub main()
   'Example of DesktopSetWallpaper
   msgbox "Click to set wallpaper to CASTLE
tiled."
   DesktopSetWallpaper "castle.bmp",true
   msgbox "Click to set wallpaper to CASTLE
centered."
   DesktopSetWallpaper "castle.bmp",false
   msgbox "Click to clear wallpaper."
   DesktopSetWallpaper "",true
end sub
```

See Also Desktop Modifications (Chapter 7)

DesktopTile Statement

Description Tiles all non-minimized top-level windows.

Syntax `DesktopTile`

Example

```
sub main()
   'Example of DesktopTile
   DesktopTile
end sub
```

See Also Desktop Modifications (Chapter 7)

Dialog Examples

OK, Cancel, and Push Buttons

```
sub main()
   'Example of a simple Dialog box
   Begin Dialog UserDialog 16,32,233,105, "Title"
      OKButton 176,7,43,14
      CancelButton 176,30,44,14
      Text 6,34,158,8, "This is a sample text field."
      PushButton 176,52,44,14, "Other #1"
      PushButton 177,76,43,14, "Other #2"
   End Dialog

   dim adialog as UserDialog
   returncode = Dialog(adialog)
   '  returncode contains the value of the button select  ed
   '  OK = -1
   '  Cancel = 0
   '  Other buttons are numbered 1 to N where N is the
number
   '  of other buttons on the dialog
   select case returncode
      case -1
         msgbox "OK pressed."
      case 0
         msgbox "Cancel pressed."
      case else
         msgbox "Other button #"+str$(returncode)+"
pressed."
   end select
end sub
```

A Comprehensive Dialog Example

```
sub main()
   'Example showing the entire process of defining,
displaying,
   'and interpreting the input from a dialog box.

   'Initialization
   '--------------
   'ListBox$ and Comb oBox are single-dimensional arrays to
hold the
   'contents of the list and combo box in the dialog box.
   Dim ListBox1$() as string
   Dim ComboBox1$() as string
```

```
   Dim stf$ as string    ' static text variable
   stf$ = "12345678901234567890123456789O"

   'In this example, the Text field uses the str$
   'variable to get its field name.  This can
   'be done for any of the dialog control types except:
   'OKButton, CancelButton, TextBox
   '
   'Note that the str$ variable must have a
   'value assigned BEFORE you can define the  dialog box.

   'Define dialog box
   '-----------------
   Begin Dialog UserDialog 16,32,304,168, "Sample Dialog
Box"
      OKButton 251,9,44,14
      CancelButton 252,30,44,14
      PushButton 252,51,44,14, "Pushbutton1"
      PushButton 252,73,44,14, "PushButton2"
      GroupBox 13,9,84,59, "Sample Group Box"
      OptionGroup .OptionGroup1
         OptionButton 21,24,65,14, "Option 1"
         OptionButton 21,44,66,14, "Option 2"
      CheckBox 15,78,79,14, "Sample Checkbox", .CheckBox1
      Text 14,105,79,8, stf$
      TextBox 16,120,81,12, .TextBox1
      ListBox 114,14,120,48, ListBox1$, .ListBox1
      ComboBox 113,68,120,84, ComboBox1$, .ComboBox1
   End Dialog
   'Prepare to display dialog box.
   '-----------------------------
   'Declare the dialog type using Dim ... as UserDialog
command
   Dim aSampleDialog as UserDialog

   'Load the list box and combo box arrays with app window
names
   AppList ComboBox1$
   AppList ListBox1$

   'Load the text box with an initial value
   aSampleDialog.TextBox1 =
"12345678901234567890123456789O"
   'Note that you reference a dialog box f ield as follows:
   '   <DialogBoxName>.<FieldName>

   'Display the Dialog
   '------------------
   a% = Dialog(aSampleDialog)   'Returns integer indicating
```

```
button chosen

   'Interpret user input
   '--------------------
   'This examples builds a string describing the contents
of the
   'dialog box when the user finished making changes.
   crlf$ = chr$(13)+chr$(10)
   dlgstr$ =  "Button pushed = "
   'Determine button pushed.
   select case a%
      case -1
         dlgstr$ = dlgstr$ + "OK"
      case 0
         dlgstr$ = dlgstr$ + "Cancel"
      case 1
         dlgstr$ = dlgstr$ + "PushButton1"
      case 2
         dlgstr$ = dlgstr$ + "PushButton2"
   end select
   dlgstr$ = dlgstr$ + crlf$
   'Determine new value of each dialog box component
   dlgstr$=dlgstr$ +  "Option = " +
str$(aSampleDialog.OptionGroup1)+crlf$
   dlgstr$=dlgstr$ +  "Checkbox = " +
str$(aSampleDialog.CheckBox1)+crlf$
   dlgstr$=dlgstr$ +  "TextBox = " + aSampleDialog.TextBox1
+ crlf$
   dlgstr$=dlgstr$ +  "ListBox = " +
ListBox1$(aSampleDialog.ListBox1)+crlf$
   dlgstr$=dlgstr$ +  "ComboBox = " +
aSampleDialog.ComboBox1
   'Display the dlgstr$ string in a message box.
   msgbox dlgstr$
end sub
```

Dialog Statement and Function

Description The `Dialog` statement displays a dialog box. The `Dialog` function displays a dialog box and returns an integer representing the button that was pushed.

Statement Syntax

```
Dialog UserDlg as dialog
```

Comments Displays the dialog box specified by the dialog template `UserDlg`.

If the user selects Cancel, a runtime error is generated. This error can be trapped using ON ERROR.

The `Dialog` statement returns only after the user presses OK, Cancel, or another push button.

Function Syntax

```
Dialog(UserDlg as dialog)
```

Returns

-1 OK button was pushed.

0 Cancel button was pushed.

>0 A push button was selected. The returned number represents which button was pushed based on its order in the dialog template. 1 represents the first button, 2 represents the second, etc.

Comments This function displays the dialog box associated with the template `UserDlg`, and returns which button was pushed. Unlike the statement form, this function will not generate a runtime error when Cancel is selected.

Example 1

```
sub main()
   'Example of using the DIALOG function and
command

   'Here is our dialog box specification.
   Begin Dialog UserDialog 16,32,148,72,
"Sample"
      OKButton 54,11,41,14
      CancelButton 55,40,41,14
   End Dialog
```

```
'Create a dialog box type variable
dim mydialog as UserDialog

'First the function method
retval% = Dialog(mydialog)
select case retval%
   case -1
      msgbox "OK pressed."
   case 0
      msgbox "Cancel  pressed."
end select
```

```
  'Now the command method
  'First create an error handler in case "Cancel"
  'is pressed.
  on error goto cancelpressed
  Dialog mydialog
  msgbox "OK pressed."
  goto endprog
cancelpressed:
  msgbox "Cancel pressed."
endprog:
  on error goto 0
end sub
```

Example 2 A Comprehensive Dialog Example (see Dialog Examples)

See Also Dialog Creation (Chapter 7)

Dim Statement

Description Declares a list of variables and their corresponding types and sizes.

A special form of the DIM statement is used to store a dialog definition in memory.

Syntax `dim name [(<subscripts>)] [as type] [,name [(<subscripts>)] [as type]]...`

Comments If a type declaration character is used (such as %, &, or $), the optional `[as type]` expression is not allowed.

Up to 60 array dimensions are allowed.

The total size of an array (not counting space for string) is limited to 42K.

Dynamic arrays are declared by not specifying any bounds:

```
dim a()
```

Any DIM statements declared within a subroutine or function are local to that subroutine or function.

If a variable is seen that has not been explicitly declared with DIM, it is implicitly declared using the type specifier character (%, $, or &). If this character is missing, then integer is assumed.

Example

```
sub main()
   'Examples of the DIM statement

   'These two statements declare integers
   dim a as integer
   dim b%
   a = 1
   b% = 1

   'These two statements declare strings
   dim s as string
   dim t$

   s = "test"
   t$ = "test"
'Array declarations
   'When an array is declared, its elements
are
   'numbered 0 through N-1 where N is the
size of
   'the array.  Using the OPTION BASE
statement,
   'you can begin numbering at 1.
dim j(5) as integer    'array of 5 integer
                          'values
   for x% = 0 to 4
      j(x%) = x%
      msgbox str$(j(x%))
   next x%
```

```
      dim k(5) as string     'array of 5
strings
   for x% = 0 to 4
      k(x%) = chr$(65+x%)
      msgbox k(x%)
   next x%

dim l(3,3) as string    'two-dimensional
array
                           'containing a
total of
                           '9 string s, 3 by 3
   for x% = 0 to 2
      for y% = 0 to 2
         l(x%,y%) = str$(x%)+str$(y%)
         msgbox l(x%,y%)
      next y%
   next x%

      'Using explicit array subscripts
   dim p(6 to 10) as integer
   for x% = 6 to 10
      p(x%) = x%
      msgbox str$(p(x%))
   next x%
end sub
```

Use with Dialogs

A special form of the DIM statement stores the preceding dialog definition (`BeginDialog...EndDialog`) in memory.

The syntax of this statement is:

```
dim dialog_name as UserDialog
```

Where `dialog_name` is a variable you define and `UserDialog` is a DCL reserved word.

For an example of this command, see Dialog Examples.

See Also

Arrays (Chapter 7)

Dialog Creation (Chapter 7)

ReDim Statement

Variables and Constants (Chapter 7)

Dir$ Function

Description Searches a disk directory.

Syntax `Dir$([filespec$])`

Returns If `filespec$` is specified, then this function returns the first file matching that filespec. If `filespec$` is not specified, then this function returns the next file matching the initial filespec.

Comments The `filespec$` argument can include wildcards, such as * and ?.

An error is generated if `dir$` is called without first calling it with a valid filespec.

If there is no matching filespec, then an empty string is returned.
If no path is specified on `filespec$`, then the current directory is used.

This function does not find hidden files and directories.

Example

```
sub main()
   'Example of Dir$() function
   a$ = dir$("c:\*.bat")   'initialize file
search
                           '(filespec parameter
required)
   do
      msgbox a$           'display file name
      a$ = dir$()         'next file name
                          '(no parameter
specified)
   loop while a$ <> ""
end sub
```

See Also File Input and Output (Chapter 7)

DirExists Function

Description Determines whether the specified `path$` exists and is a directory.

Syntax `DirExists(path$)`

Comments The function returns the integer TRUE or FALSE.

The path can be a partial path, such as "windows".

Example

```
sub main()
   'Example of DirExists
   a$ = "C:\DOS"
   if DirExists(a$) then
      msgbox a$+" exists"
   else
      msgbox a$+" does not exi st"
   end if
end sub
```

See Also File Input and Output (Chapter 7)

DiskDrives Statement

Description This statement grabs all of the valid drive letters and packs them into the specified array.

Syntax `DiskDrives list$()`

Comments The array is resized to hold the exact number of valid drives.

The `list$` parameter must be a single dimension array of strings.

Use the functions `lbound()` and `ubound()` to determine the size of the resultant array.

Example

```
sub main()
   'Example of DiskDrives command
   dim ddrives() as  string

   DiskDrives ddrives
   for i% = lbound(ddrives) to
ubound(ddrives)
      alldrives$ = alldrives$ + ddrives(i%)
   next i%
   msgbox alldrives$
end sub
```

See Also File Input and Output (Chapter 7)

DiskFree Function

Description Returns a long integer representing the free space (in bytes) available on the specified drive.

Syntax `DiskFree([drive$])`

Comments If `drive$` is empty or not specified, then the current drive is assumed.

Only the first character of the `drive$` string is used.

Example

```
sub main()
   'Example of DiskFree function
   dim ddrives() as string
   dim freespace as long

   DiskDrives ddrives
   for i% = lbound(ddrives) to
ubound(ddrives)
      if ddrives(i%) <> "A" and ddrives(i%)
<> "B" then
         freespace = DiskFree(ddrives(i%))
         msgbox "Free space on drive
"+ddrives(i%)+
                ": is "+str$(freespace)
      end if
   next i%
end sub
```

See Also File Input and Output (Chapter 7)

Do...Loop Statement

Description This statement repeats a block of DCL statements while a condition is TRUE or until a condition is TRUE.

Syntax 1

```
do {while | until} condition
  statements
loop
```

Comments Using `do while` causes the statements in the loop to be executed while the specified condition is true.

Using `do until` causes the statements in the loop to be executed until the specified condition is true.

The `condition` parameter specifies any Boolean expression.

Syntax 2

```
do
  statements
loop {while | until} condition
```

Comments Syntax 2 has the same effect as Syntax 1.

Syntax 3

```
do
  statements
loop
```

Comments If the `{while | until}` conditional clause is not specified, then the loop repeats the statements forever (or until an `exit do` statement is encountered).

Example

```
sub main()
   'Examples of DO-LOOP statements
   'All of the following loops perform the
same task.
   'The difference is primarily in WHEN the
loop exit
   'condition is checked.

   a% = 1
   do while a% = 1
      a% = msgbox("DO-WHILE-LOOP Click cancel
to go to next loop.",1)
   loop
a% = 1
   do until a% <> 1
      a% = msgbox("DO-UNTIL-LOOP Click cancel
to go to next loop.",1)
   loop

   a%=1
   do
      a% = msgbox("DO-LOOP-WHILE Click cancel
to go to next loop.",1)
   loop while a% = 1

      a% = 1
```

```
   do
      a% = msgbox("DO-LOOP-UNTIL Click cancel
to go to next loop.",1)
   loop until a% <> 1

   a% = 1
   do
      a% = msgbox("DO-LOOP Click cancel to go
to next loop.",1)
      if a% <> 1 then
         exit do
      end if
   loop
end sub
```

See Also Flow Control (Chapter 7)

DoEvents Statement

Description The `DoEvents` statement and function yield control to other applications.

Statement Syntax

```
DoEvents
```

Function Syntax

```
DoEvents[()]
```

Returns The function returns the value 0.

Comments When running in `exclusive` mode, the only way other applications can multitask is for the script to call this statement/function.

Example

```
sub main()
   'Examples of DoEvents and DoEvents()

   DoEvents
   x% = DoEvents()
end sub
```

See Also Flow Control (Chapter 7)

DoKeys Statement

Description Uses the Windows journaling mechanism to play keystrokes into the Windows environment.

Syntax `DoKeys KeyString$`

Comments The format for `KeyString$` is the same as that used for `QueKeys`.

This statement will not affect the current event queue.

Example

```
sub main()
   'Example of DoKeys
   dim alttab as string
   alttab = "%{TAB}"
   msgbox "Press OK to do first Alt-Tab"
   DoKeys alttab
   msgbox "Press OK to Alt-Tab back to
original application"
   DoKeys alttab
end sub
```

See Also Keyboard Manipulation (Chapter 7)

EditEnabled Function

Description Determines if an edit box is enabled within the current window or dialog box.

Syntax 1 `EditEnabled(name$)`

Syntax 2 `EditEnabled(ID%)`

Returns Returns the integer TRUE if the given edit box is enabled within the active window or dialog box, FALSE otherwise.

Comments If enabled, the edit box can be given the focus using the `ActivateControl` statement.

The `name$` parameter specifies the text that appears within the static control that immediately precedes the edit control in the window list or dialog template. Alternatively, the `ID%` of the edit box can be specified.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

EditExists Function

Description Determines if an edit box exists within the current window or dialog box.

Syntax 1 `EditExists(name$)`

Syntax 2 `EditExists(id%)`

Returns Returns the integer TRUE if the given edit box exists within the active window or dialog box, FALSE otherwise.

Comments If there is no active window, FALSE will be returned.

The `name$` parameter specifies the text that appears within the static control that immediately precedes the edit control in the window list (or dialog template). Alternatively, the ID of the edit box can be specified.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

EnableStopScript Statement

Description Controls whether a script may be halted by the user.

Syntax `EnableStopScript condition%`

Comments The `condition%` parameter can be one of the following constants:

- ESS_ENABLE (enable script breaking). This is the default.
- ESS_DISABLE (disable script breaking).
- ESS_ENABLE_INTERACTIVE (allows the script to be broken only when executing in interactive [debug] mode).

Example

```
sub main()
   'Example of EnableStopScript

   EnableStopScript ESS_ENABLE
      'control-break enabled
   EnableStopScript ESS_DISABLE
      'control-break disabled
   EnableStopScript ESS_INTERACTIVE
      'control-break only in script editor
end sub
```

See Also Flow Control (Chapter 7)

End Statement

Description Terminates execution of the current script.

Syntax end

Comments This statement can appear multiple times in a script, so that you can conditionally end the script. It can also appear in functions and subroutines.

All open files are close. All open DDE channels are closed.

Example

```
sub main()
   'Example of END

   msgbox "You'll see this one."
   end
   msgbox "You won't see this one."
end sub
```

See Also Flow Control (Chapter 7)

ENV_BOTH

Description Constant meaning both DOS and Windows; used by the `SetEnv` command.

ENV_DOS

Description Constant meaning DOS environment; used by the `GetEnv`, `SetEnv`, `RestoreEnv`, and `SaveEnv` commands.

ENV_WINDOWS

Description Constant meaning Windows environment; used by the `GetEnv`, `SetEnv`, `RestoreEnv`, and `SaveEnv` commands.

Environ$ Function

Description Supported only under DOS. Returns the value of the specified environment variable.

Note: For greater convenience, we recommend using the `GetEnv` function instead.

Syntax 1 `environ$(variable$)`

Syntax 2 `environ$(VariableNumber%)`

Comments If `variable$` is specified, then this function looks for that variable in the environment. If the variable name cannot be found, then an empty string is returned.

If `VariableNumber` is specified, then this function looks for the Nth variable within the environment (the first variable being number 1). If there is no such environment variables, an empty string is returned. Otherwise, the entire entry from the environment is returned in the format:

```
variable=value
```

Example

```
sub main()
   'Example of environ$() function

   a$ = environ$("path")
   msgbox a$
end sub
```

See Also Environment Statements and Functions (Chapter 7)

EOF Function

Description Determines whether end of file has been reached.

Syntax `eof(filenumber%)`

Returns Returns the integer TRUE if the end of file has been reached for the given file, otherwise FALSE.

Comments The `filenumber` parameter is a number that is used by DCL to refer to the open file—the number passed to the `open` statement.

Example Input/Output Example

See Also File Input and Output (Chapter 7)

Erl Function

Description Not used.

Syntax `erl[()]`

Returns Returns the integer 0 (DCL does not support line numbers).

See Also Error Trapping (Chapter 7)

Err Statement and Function

Description The `err` function returns an integer representing the runtime that caused the current error trap. The `err` statement sets the value returned by the `err` function to a specific value.

Statement Syntax

```
err = value%
```

Function Syntax

```
err[()]
```

Comments The `err` function can only be used while within an error trap.

When a function or statement ends, the value returned by `err` is reset to 0.

Example

```
sub main()
   'Example of Err(), Error, and Error$()
   dim i as integer

   i=1
nexterror:
   on error goto errortrap
   select case i
      case 1
         error 100
      case 2
         error 101
      case 3
         error 102
   end select
   end
errortrap:
   msgbox "Error #"+str$(err())+";
"+Error$(err())
   i = i + 1
   goto nexterror
end sub
```

See Also Error Trapping (Chapter 7)

Error Statement

Description Simulates the occurrence of the specified runtime error.

Syntax `error errornumber%`

Comments The `errornumber` parameter can be a built-in error number, or a user defined error number. The `err` function can be used within the error trap handler to determine the value of the error.

Example

```
sub main()
   'Example of Err(), Error, and Error$()
   dim i as integer

   i=1
nexterror:
   on error goto errortrap
   select case i
      case 1
         error 100
      case 2
         error 101
      case 3
         error 102
   end select
   end

errortrap:
   msgbox "Error #"+str$(err())+";
"+Error$(err())
   i = i + 1
   goto nexterror
end sub
```

See Also Error Trapping (Chapter 7)

Error$ Function

Description Returns the text corresponding to the given error number or the most recent error.

Syntax `Error$ [(errornumber%)]`

Comments If `errornumber` is omitted, then the function returns the text corresponding to the most recent runtime error. If no runtime error has occurred, then an empty string is returned ("").

If the `error` statement was used to generate a user-defined runtime error, this function will return an empty string ("").

Example

```
sub main()
   'Example of Err(), Error, and Error$()
   dim i as integer
   i=1

nexterror:
   on error goto errortrap
   select case i
      case 1
         error 100
      case 2
         error 101
      case 3
         error 102
   end select
   end

errortrap:
  msgbox "Error #"+str$(err())+"; "+Error$(err())
  i = i + 1
  goto nexterror
end sub
```

See Also Error Trapping (Chapter 7)

Exclusive Statement

Description Sets or unsets exclusive mode.

Syntax `exclusive NewState%`

Comments When set (`NewState%` is TRUE), then the script will not yield control to other applications - no other applications will execute concurrently. When a script is not running in exclusive mode (`NewState%` is FALSE), then other applications can multitask while the script is running.

By default, scripts do not run in exclusive mode.

If a script is running in exclusive mode and there is an infinite loop, you will be unable to abort the script (the computer will hang). Thus, caution must be taken to make sure that a script has no errors or infinite loops.

Scripts running exclusively run faster than scripts not in exclusive mode.

If a dialog box is encountered (`MsgBox`, `AskBox$`, `AnswerBox`, ...) while a script is running in `exclusive` mode, then that dialog box is system modal, meaning that a user can only interact with that dialog and not any other applications that may be running concurrently. All other dialog boxes, applications, and buttons will be "locked out".

Example

```
sub main()
   'Examples of the Exclusive statement

   Exclusive TRUE    'no multitasking for now
   Exclusive FALSE   'allow other programs to
run
                     'again
end sub
```

See Also Flow Control (Chapter 7)

Exit Do Statement

Description Exits a `do...loop`.

Syntax `exit do`

Comments This statement can only appear within a `do...loop` statement. It causes execution to continue with the next statement after the `loop` clause.

Example Do...Loop Statement Example

See Also Flow Control (Chapter 7)

Exit For Statement

Description Exits a `for...next` loop.

Syntax `exit for`

Comments This statement ends the `for...next` block in which it appears. Execution will continue on the line immediately after the `next` statement. This statement can only appear within a `for...next` block.

Example Exit Statement Examples

See Also Flow Control (Chapter 7)

Exit Function Statement

Description Exits the current function.

Syntax `Exit Function`

Comments This statement ends execution of the function in which it appears. Execution will continue on the statement or function following the call to this function. This statement can only appear within a function.

Example Exit Statement Examples

See Also Procedure Statements (Chapter 7)

Exit Sub Statement

Description Exits the current subroutine.

Syntax `Exit Sub`

Comments This statement ends the current subroutine. Execution is transferred to the statement following the call to the current subroutine. This statement can appear anywhere within a subroutine. It cannot appear within a function.

Example Exit Statement Examples

See Also Procedure Statements (Chapter 7)

Exit Statement Examples

```
function testfunc (a as integer) as integer
   testfunc = 0
   if a = 2 then
      Exit Function
   end if
   msgbox "no premature exit from  testfunc"
   testfunc = 100
end function

sub testsub (a as integer)
   if a = 4 then
      Exit Sub
   end if
   msgbox "no premature exit from testsub"
end sub

sub main()
   'examples of EXIT statements
   for i% = 1 to 10
      if i% >= 1 and i% <= 2 then
         b% = testfunc(i%)
         msgbox str$(b%)
      end if
      if i% >= 3 and i% <= 4 then
         testsub(i%)
      end if
      if i% = 7 then
         Exit For
      else
         msgbox "no exit yet "+str$(i%)
      end if
   next i%
   msgbox "we just exited from the for loop"
end sub
```

Exp Function

Description Returns a double-precision number representing the value of *e* raised to the power of `x`.

Syntax `Exp(x#)`

Comments Range of `x`: `0 <= x <= 709.782712893`

A runtime error is generated if `x` is out of the above specified range.

Example

```
sub main()
   'Example of EXP function

   dim result as double

   result = exp(1)
   msgbox str$(result)
end sub
```

See Also Math Statements and Functions (Chapter 7)

FALSE

Description Constant.

Returns 0

Comments Used in conditionals and Boolean expressions

FileAttr Function

Description Returns an integer representing the file mode or file handle.

Syntax `FileAttr(filenumber%,attribute%)`

Returns Returns the file mode (if `attribute` is 1), or the operating system file handle (if `attribute` is 2).

Comments If `attribute` is 1, then one of the following values is returned:

1	input
2	output
8	append

The `filenumber` parameter is a number that is used by DCL to refer to the open file—the number passed to the `open` statement.

Example Input/Output Example

See Also File Input and Output (Chapter 7)

FileCopy Function

Description Copies the specified file(s) to the given destination.

Syntax `FileCopy(src$, dest$)`

Comments The function returns the integer TRUE if successful; otherwise it returns FALSE.

Wildcards are permitted in the `src$` string. They are the same as the DOS wildcards.

If the `src$` parameter specifies wildcards, all files matching the `src$` parameter are copied to `dest$` with the proper modification made to the destination file name so that it matches the source file according to the wildcards.

Example

```
sub main()
   'Example of FileCopy

   a% = FileCopy("c:\*.bat","c:\batch")
   if a% then
      msgbox "File copy successful."
   else
      msgbox "File copy failed."
   end if
end sub
```

See Also File Input and Output (Chapter 7)

FileDateTime Function

Description Returns a double-precision number representing the date and time of the given file. The number is returned in days where Dec 20, 1899 is 0.

Syntax `FileDateTime(filename$)`

Comments This function retrieves the date and time of the file specified by `filename$`. A runtime error results if the file does not exist. The value returned can be used with the date/time functions (i.e., `year(), month(), day(), weekday(), minute(), second(), hour()`) to extract the individual elements.

Example

```
sub main()
   'Example of FileDateTime
   dim fdt as double
   dim sfdt as string

   fdt = FileDateTime("C:\AUTOEXEC.BAT")
   sfdt =
str$(month(fdt))+"/"+str$(day(fdt))+  ↵
      "/"+str$(year(fdt))+"
"+str$(hour(fdt))+ ↵

   ":"+str$(minute(fdt))+":"+str$(second(fdt)
)
   msgbox sfdt
end sub
```

See Also File Input and Output (Chapter 7)

FileDirs Statement

Description Fills an array with directory names from disk.

Syntax `FileDirs array$() [,dirspec$]`

Comments The `array$()` is any previously declared string array. The `FileDirs` function reallocates this array to exactly hold all of the directory names matching a given specification.

The `dirspec$` parameter specifies the file search mask, such as:

```
T*.    C:\*.*
```

If the `dirspec$` parameter is not specified, then `*.*` is used, which files the array with all the sub-directory names within the current directory.

Example

```
sub main()
   'Example of FileDirs statement
   dim fdirs() as string

   'list all directories on drive C:
   FileDirs fdirs,"C:\*.*"

   for i% = lbound(fdirs) to ubound(fdirs)
      fdirlist$ = fdirlist + fdirs(i%) + "; "
   next i%
   msgbox fdirlist$
end sub
```

See Also Arrays (Chapter 7)

File Input and Output (Chapter 7)

FileExists Function

Description Determines if a given filename is valid.

Syntax `FileExists(filename$)`

Returns Returns the integer TRUE if the `filename$` is a valid file, FALSE otherwise.

Example

```
sub main()
   'Example of FileExists() function

   if FileExists("C:\AUTOEXEC.BAT") then
      msgbox "Autoexec Exists."
   else
      msgbox "Autoexec Does Not Exists."
   end if
   if FileExists("D:\NADA.FIL") then
      msgbox "NADA.FIL Exists."
   else

      msgbox "NADA.FIL Does Not Exists."
   end if
end sub
```

See Also File Input and Output (Chapter 7)

FileLen Function

Description Returns a long integer representing the length of the specified file in bytes.

Syntax `FileLen(filename$)`

Comments This function is used to retrieve the length of a file without first opening the file. A runtime error results if the file does not exist.

Example

```
sub main()
   'Example of the FileLen function
   dim flen as long

   flen = FileLen("C:\AUTOEXEC.BAT")
   msgbox "Autoexec.Bat is"+str$(flen)+"
bytes long."
end sub
```

See Also File Input and Output (Chapter 7)

LOF Function

FileList Statement

Description Fills an array with filenames from disk.

Syntax `FileList array$() [,filespec$ [,fileattr%]]`

Comments The `array$()` is any previously declared string array. The `files` function reallocates this array to exactly hold all of the files matching a given filespec.

The `filespec$` parameter specifies the file search mask, such as:

```
*.EXE     *.DOC     t*.DO?
```

If the `filespec$` parameter is not specified, `*.*` is used.

The `fileattr` parameter is a number indicating what types of files you want included in the list. It can be any combination of the following:

Constant	Value	Description
ATTR_NORMAL	0	Read-only, archive, subdirectory, and files with no attributes
ATTR_READONLY	1	Read-only files
ATTR_HIDDEN	2	Hidden files
ATTR_SYSTEM	4	System files
ATTR_VOLUME	8	Volume label
ATTR_DIRECTORY	16	MS-DOS directories
ATTR_ARCHIVE	32	Files changed since last backup
ATTR_NONE	64	Files with no attributes

If the `fileattr` parameter is not specified, then the value `97` is used (`ATTR_READONLY` or `ATTR_ARCHIVE` or `ATTR_NONE` or `ATTR_DIRECTORY`). This value retrieves the same set of files normally returned by the DOS `dir` command.

Example

```
sub main()
   'Example of FileList statement
   dim ffiles() as string
   dim atrib as integer

   atrib = ATTR_NORMAL
   filelist ffiles,"C:\*.*",atrib
   for i% = lbound(ffiles) to ubound(ffiles)
      flist$ = flist$ + ffiles(i%) + "; "
   next i%
   msgbox flist$
end sub
```

See Also Arrays (Chapter 7)

File Input and Output (Chapter 7)

FileParse Statement

Description Takes a given filename and extracts a given portion of the filename from it.

Syntax `FileParse$(filename$[,operation])`

Comments The `filename$` parameter can specify an valid DOS filename (does not have to exist). For example:

```
..\TEST.DAT
C:\SHEETS\TEST.DAT
TEST.DAT
```

The optional `operation` parameter specifies which portion of the `filename$` to extract. It can be any of the following values.

```
0    full name          C:\SHEETS\TEST.DAT
1    drive              C
2    path               C:\SHEETS
3    name               TEST.DAT
4    root               TEST
5    extension          DAT
```

If `operation` is not specified, then the full name is returned. A runtime error will result if `operation` is not one of the above values.

Example

```
sub main()
   'Examples of FileParse$()
   dim filespec as string

   filespec = "C:\DOS\COMMAND.COM"

   'full file specification
   a$ = FileParse$(filespec,0)
   msgbox "Full Filespec = " + a$

   'drive
   a$ = FileParse$(filespec,1)
   msgbox "Drive = " + a$

   'path
   a$ = FileParse$(filespec,2)
   msgbox "Path = " + a$

   'name
   a$ = FileParse$(filespec,3)
   msgbox "Filename = " + a$

   'root
   a$ = FileParse$(filespec,4)
   msgbox "File Roo tname = " + a$

   'extension
   a$ = FileParse$(filespec,5)
   msgbox "File Extension = " + a$
end sub
```

See Also File Input and Output (Chapter 7)

FileType Function

Description Returns an integer representing the file type.

Syntax `FileType(filename$)`

Returns Returns one of the following file type constants:

TYPE_DOS	a DOS executable file
TYPE_WINDOWS	a Windows executable file

Comments This function is used to determine whether a file is a Windows executable or DOS executable. If one of the above values is not returned, then the file type is unknown.

Example

```
sub DisplayFileType (ftype as integer)
   select case ftype
      case TYPE_DOS
         msgbox "DOS"
      case TYPE_WINDOWS
         msgbox "WINDOWS"
      case else
         msgbox "Unknown Filetype"
   end select
end sub

sub main()
   'example of FileType function

   ftype% = FileType("C:\DOS\COMMAND.COM")
   DisplayFileType(ftype)
   ftype% = FileType("C:\DOS\DOSSHELL.EXE")
   DisplayFileType(ftype)
   ftype% =
FileType("D:\WINDOWS\NOTEPAD.EXE")
   DisplayFileType(ftype)
end sub
```

See Also File Input and Output (Chapter 7)

FindFile$ Function

Description Searches for `file$` and, if found, returns a full path to it. If the file is not found, the return value is a null-string.

Syntax `fullpath$=FindFile$(file$)`

Comments The search follows normal Windows search order: current directory, Windows directory, system directory, and then the PATH environment variable.

Example

```
sub main()
   'Example of FindFile$

   fp$ = FindFile$("notepad.exe")
   if fp$ <> "" then
      msgbox "File found as "+fp$
   else
      msgbox "File not found."
   end if
end sub
```

See Also File Input and Output (Chapter 7)

Fix Function

Description Returns the integer part of `number`.

Syntax `fix(number#)`

Comments This function returns the integer part of the given value by removing the fractional part. The sign is preserved. No rounding occurs. For example:

```
fix(4.5)       'returns 4
fix(-4.5)      'returns -4
```

Example

```
sub main()
   'Example of the Fix() function
   dim adouble as double
   dim aint as integer
   adouble = pi
   msgbox str$(adouble)
   aint = fix(adouble)
   msgbox str$(aint)
end sub
```

See Also CInt Function

Int Function

Math Statements and Functions (Chapter 7)

For...Next Statement

Description Repeats a block of statement a specified number of times, incrementing a loop counter by a given increment each time through the loop.

Syntax

```
for counter = start to end [step increment]
        ...
        ...
next [counter]
```

Comments If `increment` is not specified, then 1 is assumed.

The first time through the loop, `counter` is equal to `start`. Each time through the loop, `increment` is added to `counter` by the amount specified in `increment`.

The `for...next` statement continues executing until:

- An `exit for` statement is encountered

  OR

- When `counter` is greater than `end`.

If `end > start` then `increment` must be positive. If `end < start`, then `increment` must be negative.

The `for...next` statements can be nested. In such a case, the `next [counter]` statement applies to the innermost `for...next`.

The `next [counter]` can be optimized for next loops by separating each counter with a comma. The ordering or the counters must be consistent with the nesting order (innermost counter appearing before outermost counter):

```
next i,j
```

Example

```
sub main()
   'Examples of FOR-NEXT loops
   dim x as integer    'used as iteration
variable
                       'for FOR-NEXT
   dim xstart as integer
   dim xend as integer

   'simple form
   for x = 1 to 5
      msgbox str$(x)
   next x

   'step form
   for x = 1 to 10 step 2
      msgbox str$(x)
   next x

   'backward form
   for x = 10 to 1 step -2
      msgbox str$(x)
   next x

   'change of indexes
   for x = 69 to 74
      msgbox chr$(x)
   next x

   'variable range
   xstart = 69
   xend = 74
   for x = xstart to xend
      msgbox "VAR - "+chr$(x)
   next x

   'premature exit
   for x = 1 to 10
      if x = 6 then
         exit for
      end if
      msgbox "Exit - "+str$(x)
   next x
end sub
```

See Also Flow Control (Chapter 7)

FreeFile Function

Description Returns the next available file number.

Syntax `FreeFile[()]`

Comment The returned number is suitable for use in the open statement.

Example

```
sub main()
   'Example of FreeFile%( )
   dim nextfile as integer

   nextfile = FreeFile()
   msgbox str$(nextfile)
end sub
```

See Also File Input and Output (Chapter 7)

Function...End Function Statement

Description Creates a user-defined function.

Syntax

```
Function name[(parameter [as <type>]...)][as <type>]
  ...
  ...
  name = <expression>
End Function
```

Comments The return value is determined by the statement:

```
name = <expression>
```

The name of the function following DCL naming conventions. It can include type declaration characters: %, &, and $.

If no assignment is encountered before the function exits, then 0 will be returned for numeric functions, and an empty string will be returned for string functions.

The type of the return value is determined by the `as <type>` clause on the function statement itself. As an alternative, a type declaration character can be added to the function name:

```
Function Test() as string
  Test = "Hello World"
End Function

Function Test$()
  Test = "Hello World"
End Function
```

Parameters are passed to a function by reference, meaning that any modifications to a passed parameter changes that variable in the caller. To avoid this, simply enclose variable names in parenthesis, as in the following example function calls:

```
i = UserFunction(10,12,(j))
```

If a function is not to receive a parameter by reference, then the optional `byval` keyword can be used:

```
Function Test(byval FileName as string)
       as string
End Function
```

A function returns to the caller when either of the following statements is encountered:

```
End Function
Exit Function
```

The function definition must precede the statements that call the function.

The function cannot be inside a subroutine (including `sub(main)`) or inside another function.

Functions can be recursive.

See Also Procedure Statements (Chapter 7)

GetAttr Function

Description Returns an integer containing the attributes of the specified file.

Syntax `GetAttr(filename$)`

Returns The attribute value returned contains the sum of the following attributes.

Constant	Value	Description
ATTR_NORMAL	0	Read-only, archive, subdirectory, and files with no attributes
ATTR_READONLY	1	Read-only files
ATTR_HIDDEN	2	Hidden files
ATTR_SYSTEM	4	System files
ATTR_VOLUME	8	Volume label
ATTR_DIRECTORY	16	MS-DOS directories
ATTR_ARCHIVE	32	Files changed since last backup
ATTR_NONE	64	Files with no attributes

These attributes are the same as those used by DOS.

Example

```
sub main()
   'Example of GetAttr

   i% = GetAttr("C:\IO.SYS")
   msgbox str$(i%)
end sub
```

See Also File Input and Output (Chapter 7)

GetCheckbox Function

Description Returns an integer representing the state of the specified check box.

Syntax 1 `GetCheckbox(name$)`

Syntax 2 `GetCheckbox(id%)`

Comments This function is used to determine the state of a check box, given its `name$` (the text of its label) or `id%`. The return value will be one of the following:

0 check box has no check
1 check box contains a check
2 check box is grayed

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

GetComboboxItem$ Function

Description Returns the text corresponding to an item number in a combobox.

Syntax 1 `GetComboboxItem$(name$,ItemNumber%)`

Syntax 2 `GetComboboxItem$(id%,ItemNumber$)`

Comments The combobox must exists within the current window or dialog box, otherwise a runtime error is generated.

You can use the `name$` or `id%` parameter to specify the combobox. The `name$` parameter specifies the text that appears in the static control that immediately precedes the combobox control in the window list (or dialog template).

The `ItemNumber%` parameter is the line number of the desired combobox item.

An empty string will be returned if the combobox does not contain textual items.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

GetComboboxItemCount Function

Description Returns an integer representing the number of items in the specified combobox.

Syntax 1 `GetComboBoxItemCount(name$)`

Syntax 2 `GetComboBoxItemCount(id%)`

Comments You can use the `name$` or `id%` parameter to specify the combobox. The `name$` parameter specifies the text that appears in the static control that immediately precedes the combobox control in the window list (or dialog template).

A runtime error is generated if the specified combobox does not exists within the current window or dialog box.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

GetEditText$ Function

Description Returns the textual content of the specified edit box.

Syntax 1 `GetEditText$(name$)`

Syntax 2 `GetEditText$(id%)`

Comments The name of an edit control is determined by scanning the window list (or dialog template) for a static control labeled `name$` that is immediately followed by an edit control. A runtime error is generated if an edit control with that name cannot be found within the active window.

For edit controls that do not have a preceding static control, the `id$` can be used to absolutely reference the control. The id% is assigned to the control by the program. It can be obtained through Windows diagnostic utilities, such as SPY.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

GetEnv Function

Description Returns the value of the given environment variable for DOS or Windows.

Syntax `GetEnv$(var$[, mode$])`

Comments The `mode$` parameter can be ENV_DOS or ENV_WINDOWS.

If `mode$` is ENV_DOS, the variable is returned from the DOS environment, otherwise it is returned from the Windows environment.

If mode is unspecified, the default is ENV_WINDOWS.

Example

```
sub main()
   'Example of SetEnv and GetEnv$()
   tmp$ = GetEnv$("TMP",ENV_WINDOWS)
   tv$ = AskBox$("New Value For TMP
Environment Variable:")
   a% = SetEnv("TMP",tv$,ENV_WINDOWS)
   msgbox "New value for TMP is "+
GetEnv$("TMP",ENV_WINDOWS)
   'restore old value
   a% = SetEnv("TMP",tmp$,ENV_WINDOWS)
end sub
```

See Also Environment Statements and Functions (Chapter 7)

GetListboxItem$ Function

Description Returns the string corresponding to a list box item.

Syntax 1 `GetListboxItem$(name$,item%)`

Syntax 2 `GetListboxItem$(id%,item%)`

Comments This function retrieves the text of a given item in a listbox. The `item%` parameter is the item's position in the list, where 1 is the first item. The `item%` parameter must be between 1 and the number of items in the listbox.

The listbox can be specified using either its `id%` or the `name$` (label) of the static control that immediately precedes the listbox control in the window list (or dialog template).

A runtime error is generated if the specified listbox cannot be found within the active window.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

GetListboxItemCount Function

Description Returns an integer representing the number of items in the specified listbox.

Syntax 1 `GetListboxItemCount(name$)`

Syntax 2 `GetListboxItemCount(id%)`

Comments The listbox can be specified using either its `id%` or the `name$` (label) of the static control that immediately precedes the listbox control in the window list (or dialog template).

A runtime error is generated if the specified listbox cannot be found within the active window.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

GetOption Function

Description Determines whether a given option button is checked.

Syntax 1 `GetOption(name$)`

Syntax 2 `GetOption(id%)`

Returns Returns the integer TRUE if the option is set, FALSE otherwise.

Comments The option button must exist within the current window or dialog box.

The option button can be referenced given its `name$` (label) or its `id%`. A runtime error will be generated if the given option button does not exist.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

GetUserName Function

Description Use the `NetUserName` function instead.

GoSub Statement

Description Executes the specified subroutine.

Syntax `Gosub` *`label`*

Comments This statement causes execution to continue at the specified label. Execution can later be returned to the statement following the `gosub` by using the `return` statement.

The `label` parameter must be a label within the current function or subroutine. The `gosub` statement outside the context of the current function or subroutine is not allowed.

Note: It is a sounder programming practice to write a named subroutine using a `Sub...End Sub` block.

Example

```
sub main()
   'Examples of GOSUB and RETURN

   for x% = 1 to 5
      Gosub mylabel
   next x%
   end

mylabel:
   msgbox "Here we are!"
   return
end sub
```

See Also Flow Control (Chapter 7)

Goto Statement

Description Transfers execution to the line containing the specified label.

Syntax `Goto <label>`

Comments The compiler will produce an error if `label` does not exist.

The `label` must appear within the same subroutine or function as the `goto`.

Labels must begin with a letter and end with a colon. Keywords cannot be used as labels. Labels are not case sensitive.

Example

```
sub main()
   'Example of GOTO

   for x% = 1 to 5
      if x% = 3 then goto enditall
   next x%
   msgbox "Error"
   end

enditall:
   msgbox "Ended properly"
   end
end sub
```

See Also Flow Control (Chapter 7)

GroupBox Statement

Description Defines a groupbox within a dialog box template.

Syntax `GroupBox x%,y%,width%,height%,title$`

Comments A groupbox is simply a visual element used to enclose other controls within a dialog box.

This statement can only appear within a dialog box template definition (BEGIN DIALOG...END DIALOG).

The `x%,y%,width%,height%` parameters are specified in dialog coordinates. The `x,y` position is relative to the upper left corner of the dialog box.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

Hex$ Function

Description Returns a string containing the hexadecimal equivalent of the specified number.

Syntax `hex$(number&)`

Comments The returned string contains only the number of hexadecimal digits necessary to represent the number, up to a maximum of 8.

The `number` parameter can be any type, but is rounded to the nearest whole number before converting to hexadecimal. If the passed number is an integer, then a maximum of 4 digits are returned; otherwise, up to 8 digits can be returned.

Example

```
sub main()
   'Example of Hex$() function

   i% = 31
   h$ = hex$(i%)
   msgbox h$
end sub
```

See Also Conversions (Chapter 7)

HLine Statement

Description Scrolls the window with the focus left or right by the specified number of lines. This feature is useful when the contents are wider than the window.

Syntax `HLine [lines%]`

Comments If the `lines` parameter is omitted, then the window is scrolled right by 1 line.

Example

```
sub main()
   'Examples of HLINE

   ViewPortOpen
   ViewPortClear
   Print "Here is some test data."
   HLine 50
   sleep 2000
   HLine -50
   ViewPortClose
end sub
```

See Also Window Manipulation (Chapter 7)

Hour Function

Description Returns an integer representing the hour of the day encoded in the specified `serial` parameter. The value returned is between 0 and 23 inclusive.

Syntax `hour(serial#)`

Comments You can obtain the value for the `serial#` parameter by using the `TimeSerial` or `TimeValue` command.

Example

```
sub main()
   'Example of hour() function
   dim dt as double

   dt = Now
   msgbox str$(hour(dt))    'current hour
end sub
```

See Also Date and Time Functions (Chapter 7)

HPage Statement

Description Scrolls the window with the focus left or right by the specified number of pages. This feature is useful when the contents are wider than the window.

Syntax `HPage [pages%]`

Comments If the `pages%` parameter is omitted, then the window is scrolled right by 1 page.

Example

```
sub main()
   'Examples of HPage

   ViewPortOpen
   ViewPortClear
   Print "This is some test data"
   HPage 1
   sleep 2000
   HPage -1
   ViewPortClose
end sub
```

See Also Window Manipulation (Chapter 7)

HScroll Statement

Description Sets the thumb mark on the horizontal scroll bar attached to the current window.

Syntax `HScroll percentage%`

Comments The position is given as a percentage of the total range associated with that scroll bar. For example, if the `percentage%` parameter is 50, then the thumb is positioned in the middle of the scroll bar.

Example

```
sub main()
   'Example of HSCROLL

   ViewPortOpen
   ViewPortClear
   Print "This is some test data for the
viewport scroll test."
   sleep 2000

   HScroll 50  '50 percent scroll
   sleep 2000
   HScroll 0      'no scroll
   sleep 2000
   ViewPortClose
end sub
```

See Also Window Manipulation (Chapter 7)

If...Then...Else Statement

Description Conditionally executes a statement or group of statements.

Syntax 1

```
if <condition> then <statement>
[else <statement>]
```

Syntax 2

```
if <condition> then
  [<statement>]
[elseif <condition> then
  [<statement>]]
[else
  [<statement>]]
end if
```

Comments In the single line version, the `<statement>` must be a single statement. Optionally, many statements can be separated using the colon (:).

Example

```
sub main()
   'Example IF-THEN-ELSE-END IF statement
   dim a%

   if a% = 1 then
      'do this stuff if a is 1
   elseif a% = 2 then
      'otherwise if a is 2 then do this stuff
   else
      'if a is neither 1 nor 2 then do this
stuff
   end if
end sub
```

See Also Flow Control (Chapter 7)

Input # Statement

Description Reads comma-delimited data from a file into variables.

Syntax `Input [#]filenumber%,variable[,variable]...`

Comments This statement reads data from the file referenced by `filenumber%` into the given variables.

Each `variable` must be type matched to the data in the file. For example, a string variable must be matched to a string in the file.

All data items are separated by commas in the file. Leading spaces are ignored. Strings must be enclosed in quotes:

```
10,"Hello World",192,6
```

The `filenumber%` parameter is a number that is used by DCL to refer to the open file—the number passed to the `open` statement.

The `filenumber%` must reference a file opened in `input` mode.

See Also File Input and Output (Chapter 7)

Input$ Function

Description Returns a character string containing the first `numbytes%` characters read from the given file.

Syntax `input$(numbytes%,[#]filenumber%)`

Comments The `input$` function reads all characters, including spaces and carriage returns.

The `filenumber%` must reference a file opened in `input` mode.

The `filenumber%` parameter is a number that is used by DCL to refer to the open file—the number passed to the `Open` statement.

See Also File Input and Output (Chapter 7)

Input/Output Example

```
sub main()
   'Example of file input/output functions
   'Open, Close, Eof, FileAttr, Line Input#, Lof
   'Because of the use of ViewPort... commands,
   'this script should not be run under DOS.

   dim fileno as integer
   dim flen as integer
```

```
   dim fileatr as integer
   dim osfile as integer
   dim inline as string
   dim linedesc as string
   dim linemult as integer

   ViewPortOpen "AUTOEXEC.BAT"
   ViewPortClear
   fileno = FreeFile()   'get next available file number
   Open "C:\AUTOEXEC.BAT" for input as fileno
   flen = Lof(fileno)   'get length of file
   Print "File Autoexec.Bat - Length"+str$(flen)
   fileatr = FileAttr(fileno,1)
   osfile = FileAttr(fileno,2)
   Print "File is opened for ";
   Select Case fileatr
      Case 1
         Print "Input";
      Case 2
         Print "Output";
      Case 8
         Print "Append";
   End Select
   Print " and has an Operating System Filehandle
of"+str$(osfile)
   Print string$(50,"-")
   while not eof(fileno)
      Line Input #fileno, inline
      Print inline
   wend
   Close fileno
   msgbox "Click OK when you're finished viewing the file  ."
   ViewPortClear
   Print "Creating a simple output file."
   fileno = FreeFile()
   Open "C:\JUNK.TXT" for output as fileno
   for i% = 1 to 10
      Write #fileno,"Line"+str$(i%),i% * 10
   next i%
   Close fileno
   Print "Reading created file."
   Print string$(50,"-")
   fileno = FreeFile()
   Open "C:\JUNK.TXT" for input as fileno
   while not eof(fileno)
      print "Seek Position"+str$(Seek(fileno))
      print "File Position"+str$(loc(fileno))    'print the
file position
      Input #fileno,linedesc,linemult
      Print linedesc+", Value ="+ str$(linemult)
```

```
   wend
   Close fileno
   msgbox "Click OK when you're finished viewing the file."
   ViewPortClear
   Print "Display Autoexec.Bat again...this time in a
different way."
   fileno = FreeFile()
   Open "C:\AUTOEXEC.BAT" for input as fileno
   flen = lof(fileno)
   aexec$ = Input$(flen,fileno)   'read the entire file at
once
   print aexec$
   Close fileno
   msgbox "Click OK when you're finished viewing the file."
   ViewPortClear
   Print "Now create Junk.Txt using Print# instead of
Write#"
   fileno = FreeFile()
   Open "C:\JUNK.TXT" for output as fileno
   for i% = 1 to 10
      Print #fileno,i%,"test","more"
   next i%
   Close fileno
   fileno = FreeFile()
   Open "C:\JUNK.TXT" for input as fileno
   stuff$ = Input$(lof(fileno),fileno)
   Close fileno
   print stuff$
   msgbox "Click OK when you're finished viewing the file."
   ViewPortClose
end sub
```

InputBox$ Function

Description Presents a dialog box displaying a prompt and returns the user's response.

Syntax `InputBox$(prompt$ [,title$ [,default$ [,x%,y%]]])`

Returns Returns the text contained in the edit box when the user presses OK. If the user Cancels the dialog box, an empty string is returned.

Comments A default response can be specified in the `default$` parameter.

The `prompt$` parameter can contain multiple lines, each separated with a Carriage Return/Line Feed (`chr$(13) + chr$(10)`).

The `title$` parameter specifies the text that appears in the dialog box's caption. If the `title$` parameter is not specified, no title appears in the dialog's caption.

The `x` and `y` parameters are specified in twips (1/20th of a point or 1/1440 of an inch). This allows the dialog box to be positioned in a device independent manner. If the position is not specified, then the dialog box is positioned on or near the object containing the executing script.

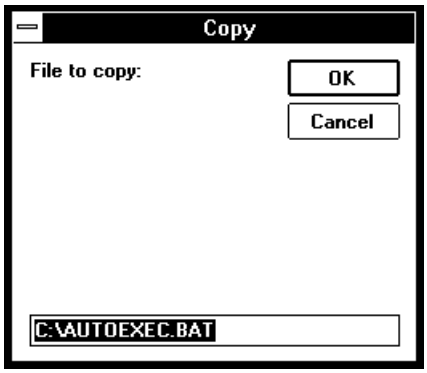
Example 1

```
sub main()
   'Example of InputBox$() function

   a$ = InputBox$("Enter your description:",
"Description","",100,200)
   msgbox a$
end sub
```

Example 2

```
s$ = InputBox$("File to
copy:","Copy","C:\AUTOEXEC.BAT")
```



See Also Dialog Display (Chapter 7)

InStr Function

Description Searches a string for a substring.

Syntax `instr([start%,] search$,find$)`

Returns Returns an integer representing the character position of `find$` within `search$`.

Comments If the string is found, its character position within `search$` is returned, with 1 being the character position of the first character.

If `start%` is specified, then the search starts at that character position within `search$`. The `start%` parameter must be between 1 and 65535. If not specified, the search starts at the beginning (`start% = 1`).

If the string is not found, or `start%` is greater than the length of `search$`, or if `search$` is empty, then 0 is returned.

Example

```
sub main()
   'Example of InStr() function
   dim teststring as string

   teststring = "The quick brown fox jumps
over the ↵
      lazy dog."
   'search starting at position 1
   i% = InStr(1,teststring,"quick")
   if i% = 0 then
      msgbox "'quick' was not found"
   else
      msgbox "'quick' was found at
position"+str$(i%)
   end if
   'search starting at positio n 10
   i% = InStr(10,teststring,"quick")
   if i% = 0 then
      msgbox "'quick' was not found"
   else
      msgbox "'quick' was found at
position"+str$(i%)
   end if
end sub
```

See Also Strings (Chapter 7)

Int Function

Description Returns the integer part of a given number.

Syntax `int(number#)`

Comments This function returns the first integer less than (rounds down) `number#`. The sign is preserved.

Example

```
sub main()
   'example of INT function
   dim adouble as double

   adouble = pi
   msgbox str$(adouble)
   msgbox str$(int(adouble))
end sub
```

See Also CInt Function

Fix Function

Math Statements and Functions (Chapter 7)

Item$ Function

Description Gets a set of contiguous, delimited items from a text string.

Syntax `item$(text$,first%,last% [,delimiters$])`

Returns Returns all of the items between `first%` and `last%` within the specified text.

Comments The `first%` parameter specifies the first item in the sequence to return. The lowest value for `first%` is 1. All items between `first%` and `last%` are returned.

By default, items are separated by commas and end-of-lines. This can be changed by specifying different delimiters in the `delimiters$` parameter.

If `first` is greater than the number of items in `text$`, then an empty string is returned.

If `last` is greater than the number of items in `text$`, then all items from `first` to the end of text are returned.

Example

```
sub main()
   'Example of Item$() and ItemCount
   dim pathstr as string

   pathstr = environ$("PATH")     'get the
path
   msgbox "There
are"+str$(itemcount(pathstr,";"))+" items in
the path."
   msgbox pathstr
   msgbox item$(pathstr,3,4,";")
end sub
```

See Also Strings (Chapter 7)

ItemCount Function

Description Returns an integer representing the number of items in the specified text.

Syntax `ItemCount(text$ [,delimiters$])`

Comments By default, items are separated by commas and end-of-lines. This can be changed by specifying different delimiters in the `delimiters$` parameter. For example, to parse items using a backslash:

```
n = itemCount(text$,"\")
```

The first `text$` item is 1.

Example Item$ Example

See Also Strings (Chapter 7)

Kill Statement

Description Deletes one or more files.

Syntax `kill filespec$`

Comments This command deletes all files matching `filespec$`.

The `filespec$` parameter can contain wildcards, such as * and ?.

This function behaves the same as the "del" command in DOS.

Example

```
sub main()
   'Example of KILL command

   kill "c:\junk.txt"
end sub
```

See Also File Input and Output (Chapter 7)

LBound Function

Description Determines the smallest subscript for a dimension of an array.

Syntax `lbound(ArrayVariable() [,dimension%])`

Returns Returns an integer representing the lower bound of the specified dimension of the specified array variable. If the array has no dimension, a runtime error is returned.

Comments The first dimension is assumed if `dimension%` is not specified (i.e., `dimension% = 1`).

Example

```
sub main()
   'Examples of LBOUND
   dim ia1(8) as integer
   dim ia2(65 to 70) as integer

   msgbox str$(lbound(ia1))
   msgbox str$(lbound(ia2))
end sub
```

See Also Arrays (Chapter 7)

LCase$ Function

Description Returns the lower case equivalent of the specified string.

Syntax `Lcase$(str)`

Example

```
sub main()
   'Example of LCase$
   dim teststr as string

   teststr = "THIS IS A TEST"
   msgbox teststr
   teststr = Lcase$(teststr)
   msgbox teststr
end sub
```

See Also Strings (Chapter 7)

Left$ Function

Description Returns the leftmost `NumChars%` characters from a given string.

Syntax `Left$(str$,NumChars%)`

Comments If `NumChars%` is 0, then an empty string is returned.

If `NumChars%` is greater than or equal to the number of characters in the specified string, then the entire string is returned.

Example

```
sub main()
   'Example of left$()
   dim teststr as string

   teststr = "This is a test."
   msgbox teststr
   teststr = left$(teststr,7)
   msgbox teststr
end sub
```

See Also Strings (Chapter 7)

Len Function

Description Returns an integer representing the number of characters in a given string.

Syntax `Len(str$)`

Comments If `str` is empty, 0 is returned.

Example

```
sub main()
   'Example of Len()
   dim teststr as string

   teststr = "This is a test."
   msgbox "'"+teststr+"'
is"+str$(len(teststr))+" characters long."
end sub
```

See Also Strings (Chapter 7)

Let Statement

Description Assigns the result of an expression to a variable.

Syntax `[Let] variable = expression`

Comments The `let` statement is supported for compatibility with other implementations of DCL.

Example

```
sub main()
   'Examples of Let statement

   let a% = 1
   let s$ = "test"
end sub
```

See Also Variables and Constants (Chapter 7)

Line$ Function

Description Gets a set of lines (delimited by CR/LFs) from the specified text.

Syntax `Line$(text$,first%[,last%)`

Returns Returns a single line or group of lines between `first` and `last`.

Comments Lines are delimited by CR/LF pairs.

If `last` is not specified, then only one line is returned.

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$`, all lines from `first` to the end of text are returned.

Example

```
sub main()
   'Example of Line$()
   dim crlf as string
   dim testlines as string

   crlf = chr$(13) + chr$(10)
      'carriage return followed by a linefeed
   testlines = "line 1" + crlf
   testlines = testlines + "line 2" + crlf
   testlines = testlines + "line 3" + crlf
   testlines = testlines + "line 4"
   msgbox testlines
   msgbox line$(testlines,2,3)
end sub
```

See Also Strings (Chapter 7)

LineCount Function

Description Returns an integer representing the number of lines in the specified text.

Syntax `LineCount(text$)`

Comments Lines are delimited by CR/LF pairs.

Example

```
sub main()
   'Example of LineCount
   dim crlf as string
   dim testlines as string
   crlf = chr$(13) + chr$(10)
      'carriage return followed by a linefeed
   testlines = "line 1" + crlf
   testlines = testlines + "line 2" + crlf
   testlines = testlines + "line 3" + crlf
   testlines = testlines + "line 4"
   msgbox testlines
   msgbox str$(linecount(testlines))+" lines
total."
end sub
```

See Also Strings (Chapter 7)

LineInput # Statement

Description Reads a line into a string variable.

Syntax `LineInput [#]filenumber%,text$`

Comments This statement reads an entire line into the given string variable `text$`. The file is read up to the next carriage return. The file pointer is positioned after the terminating CR/LF.

The `filenumber%` parameter is a number that is used by DCL to refer to the open file—the number passed to the `open` statement.

The `filenumber%` must reference a file opened in `input` mode.

The `text$` parameter is any string variable reference.

Example Input/Output Example

See Also File Input and Output (Chapter 7)

ListBox Statement

Description Defines a listbox that appears within a dialog box template.

Syntax `ListBox x%,y%,width%,height%,items$(),.Field`

Comments The `items$` array must be a single-dimension array of strings. The elements of this array are placed into the listbox when the dialog box is created. The `.Field` parameter defines the name used to extract which string occupies the listbox when the dialog box ends. On exit from the `Dialog` statement, the `.Field` will contains an index to the item that is highlighted in the listbox.

This statement can only appear within a dialog box template definition (BEGIN DIALOG...END DIALOG).

The `x%,y%,width%,height%` parameters are specified in dialog coordinates. The `x,y` position is relative to the upper left corner of the dialog box.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

ListboxEnabled Function

Description Determines whether a listbox is enabled within the current window or dialog box.

Syntax `ListboxEnabled(name$ | id%)`

Returns Returns the integer TRUE if the given listbox is enabled within the active window or dialog box, FALSE otherwise.

Comments If there is no active window, FALSE is returned.

The `name$` parameter specifies the text that appears within the static control that immediately precedes the listbox control in the window list (or dialog template). Alternatively, the `id%` of the listbox can be specified.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

ListboxExists Function

Description Determines whether a listbox exists within the current window or dialog box.

Syntax `ListboxExists(name$ | id%)`

Returns Returns the integer TRUE if the given listbox exists within the active window or dialog box, FALSE otherwise.

Comments If there is no active window, FALSE is returned.

The `name$` parameter specifies the text that appears within the static control that immediately precedes the listbox control in the window list (or dialog template). Alternatively, the `id%` of the listbox can be specified.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

Loc Function

Description Returns an integer representing the position of the file pointer in the given file.

Note: We recommend using the `Seek` function instead.

Syntax `Loc(filenumber%)`

Comments The `filenumber%` parameter is a number that is used by DCL to refer to the open file—the number passed to the `open` statement.

See Also File Input and Output (Chapter 7)

LOF Function

Description Returns an integer representing the number of bytes in the given file.

Syntax `LOF(filenumber%)`

Comments The `filenumber` parameter is a number that is used by DCL to refer to the open file—the number passed to the `open` statement.

Example Input/Output Example

See Also File Input and Output (Chapter 7)

Log Function

Description Returns a double-precision number representing the natural logarithm of a given number.

Syntax `Log(number#)`

Comments The value of `number` must be greater than 0.

Example

```
sub main()
   'Example of log() function

   msgbox "Log of 1 is "+str$(L og(1))
   msgbox "Log of 2 is "+str$(Log(2))
end sub
```

See Also Math Statements and Functions (Chapter 7)

LTrim$ Function

Description Returns the specified string with the leading spaces removed.

Syntax `Ltrim$(str$)`

Example

```
sub main()
   'Example of LTrim$()
   dim teststring as string

   teststring = "     testing     "
   msgbox "*" + teststring + "*"
   msgbox "*" + Ltrim$(teststring) + "*"
end sub
```

See Also Strings (Chapter 7)

Main Statement

Description Defines the Main subroutine for the script.

Syntax
```
sub main()
end sub
```

Comments This defines the subroutine that receives execution control from the host application.

See Also Procedure Statements (Chapter 7)

MCI Function

Description Executes MCI (multimedia) command.

Syntax `MCI(command$,result$ [,error$])`

Returns Returns the integer 0 if the function was successful, otherwise it returns an error number.

Comments If an error occurs, then the optional `error$` parameter is set to the text corresponding to the error.

If the `command$` returns a value, then that value is contained in `result$`.

The `mci` function accepts any MCI command as defined in the *Multimedia Programmers Reference* in the Windows 3.1 SDK.

Example
```
sub main()
   'Example MCI command
   'NOTE:  This program may not run on your
machine.

   dim resultstr as string
   dim errorstr as string

   MCI("play cdrom from 1 to
20",resultstr,errorstr)
end sub
```

See Also Environment Statements and Functions (Chapter 7)

Menu Statement

Description Issues the specified menu command from the active window.

Syntax `Menu MenuItem$`

Comments The `MenuItem$` parameter specifies the complete menu item name, each menu level separated with a period. For example, the `"Open"` command on the `"File"` menu is represented by: `"File.Open"`. Cascading menu items may have multiple periods, one for each popup menu, such as `"File.Layout.Vertical"` . Menu items can also be specified using numeric index values. For example, to select the third menu item from the File menu, use `"File.#3"` To select the 4th item from the third menu, use `"#3,#4"`. Separators counts as items.

Items from an application's system menu can be selected by beginning the menu item specification with a period, such as `".Restore"` or `".Minimize"`.

A runtime error will result if the menu item specification does not specify a menu item. For example, `"File"` specifies a menu popup, rather than a menu item. The menu item `"File.Blank Blank"` is not a valid menu item.

When comparing menu item names, this statement removes periods (.), spaces, and &. Further, all characters after a backspace or tab are removed. Thus, the menu item "&Open...\aCtrl+F12" translates simply to "Open".

A runtime error is generated if the menu item cannot be found or is not enabled at the time that this statement is encountered.

Example

```
sub main()
   'Example of Menu command

   aname$ = AppFind$("Notepad")
   AppActivate aname$
   Menu "File.Page Setup"
end sub
```

See Also Menus (Chapter 7)

MenuItemChecked Function

Description Determines whether a menu item in the active window is checked.

Syntax `MenuItemChecked(MenuItemName$)`

Returns Returns the integer TRUE if the given menu item exists and is checked, FALSE otherwise.

Comments The `MenuItemName$` parameter specifies a complete menu item or menu item popup following the same format as that used by the `Menu` statement.

See Also Menus (Chapter 7)

MenuItemEnabled Function

Description Determines whether a menu item in the active window is enabled.

Syntax `MenuItemEnabled(MenuItemName$)`

Returns Returns the integer TRUE if the given menu item exists and is enabled, FALSE otherwise.

Comments The `MenuItemName$` parameter specifies a complete menu item or menu item popup following the same format as that used by the `Menu` statement.

See Also Menus (Chapter 7)

MenuItemExists Function

Description Determines whether a menu item in the active window exists.

Syntax `MenuItemExists(MenuItemName$)`

Returns Returns the integer TRUE if the given menu item exists, FALSE otherwise.

Comments The `MenuItemName$` parameter specifies a complete menu item or menu item popup following the same format as that used by the `Menu` statement.

Examples

```
sub main()
if MenuItemExists("File.Open") then beep
if MenuItemExists("File") then
   MsgBox "There is a File menu."
end sub
```

See Also Menus (Chapter 7)

Mid$ Function

Description The `mid$` function returns a substring. The `mid$` function can also replace a substring with new text.

Syntax 1 `Mid$(str$,start% [,length%])`

Returns Returns a substring from the specified string. The substring starts at character position `start%` for `length%` number of characters.

Comments If `length%` is not specified, then the entire string starting at `start%` is returned.

If `start%` is greater than the length of `str$`, then an empty string is returned.

Syntax 2 `Mid$(str$,start%[,length%]) = newvalue$`

Comments This statement replaces one part of a string with another. The `str$` parameter specifies the string variable containing the substring to replace. The substring within `str$` which is replaced starts at the character position specified by `start%` for `length%` number of characters. If `length%` is not specified, then the rest of the string is assumed.

The `newvalue$` parameter is any string or string expression. The resultant string is never longer than the original length of `str$`. Any extra characters at the end of `newvalue$` are ignored.

Example

```
sub main()
   'Example of mid$() function and command
   dim teststr as string

   teststr = "This is a test."
   msgbox teststr
   msgbox mid$(teststr,3,5)    'starting at 3
and
                               'retrieving 5
characters
   mid$(teststr,3,5) = "     "
   msgbox teststr
   msgbox mid$(teststr,3,5)    'starting at 3
and
                               'retrieving 5
characters
end sub
```

See Also Strings (Chapter 7)

Minute Function

Description Returns an integer representing the minute of the day encoded in the specified `serial#` parameter. The value returned is between 0 and 59 inclusive.

Syntax `Minute(serial#)`

Comment You can obtain the value for the `serial#` parameter by using the `TimeSerial` or `TimeValue` command.

Example

```
sub main()
   'Example of minute() function
   dim dt as double

   dt = Now
   msgbox str$(minute(dt))  'current minute
end sub
```

See Also Date and Time Functions (Chapter 7)

MkDir Statement

Description Creates a new directory.

Syntax `MkDir dir$`

Comments This command behaves just like the DOS "md" command.

Example

```
sub main()
   'Example of mkdir

   MkDir "C:\testdir"
end sub
```

See Also File Input and Output (Chapter 7)

Mod

Description Modulo operator.

Syntax `expression1 mod expression2`

Returns Returns the remainder of `expression1 / expression2` .

Notes The two operands are converted to whole numbers before performing the modulo operation.

Example

```
sub main()
   'Example of MOD

   msgbox str$(5 mod 3)    'should display 2-
-the
                           'remainder after
division
end sub
```

See Also Operators (Chapter 7)

Month Function

Description Returns an integer representing the month of the date encoded in the specified `serial#` parameter. The value returned is between 1 and 12 inclusive.

Syntax `month(serial#)`

Comment You can obtain the value for the `serial#` parameter by using the `DateSerial` and `DateValue` command.

Example

```
sub main()
   'Example of Month

   msgbox str$(month(Now))    'display
current month
end sub
```

See Also Date and Time Functions (Chapter 7)

Message Example

```
sub main()
   'Example of MsgOpen, MsgSetText, MsgSetThermometer, and
MsgClose
   MsgOpen "Message Box - 0% complete.",0,TRUE,TRUE
   on error goto cancelpressed
   for i% = 1 to 100
      sleep 100
      msg$ = "Message Box -"+str$(i%)+"% complete."
      MsgSetText msg$
      MsgSetThermometer I%
   next I%
   MsgClose
   msgbox "Finished Normally"
   end

cancelpressed:
   MsgClose
   on error goto 0
   msgbox "Cancel was selected."
   end
end sub
```

MsgBox Statement and Function

Description The `MsgBox` statement displays a messages box; the `MsgBox` function displays a message box and returns an integer representing the button that was pressed.

Statement Syntax

```
MsgBox msg$ [,type% [,title$]]
```

Function Syntax

```
MsgBox(msg$ [,type% [,title$]])
```

Returns The `MsgBox` function returns one of the following numbers:

1 OK was pressed

2 Cancel was pressed

3 Abort was pressed

4 Retry was pressed

5 Ignore was pressed

6 Yes was pressed

7 No was pressed

Comments The dialog box is sized to hold the entire contents of `msg$`. The `msg$` string can contain CR/LF to separate lines. If a given line is too long, it will be word wrapped.

The `type` parameter is the sub of the any of the following values:

0 display OK button only

1 display OK, Cancel buttons

2 display Abort, Retry, Ignore buttons

3 display Yes, No, Cancel buttons

4 display Yes, No buttons

5 display Retry, Cancel buttons

16 display "stop" icon 

32 display "question mark" icon 

48 display "exclamation point" icon 

64 display "information" icon 

0 first button is the default button

256 second button is the default button

512 third button is the default button

0 Application modal - the current application is suspended until dialog box is closed

4096 System modal - all applications are suspended until the dialog box is closed

The default value for `type` is 0 (display only the OK button, making it the default).

The default value for `title$` is "DCL".

Example

```
sub main()
   'Examples of MsgBox function and statement

   for i% = 0 to 5
      MsgBox "message", i%, "title"
   next i%
   for i% = 0 to 5
      result = MsgBox("message", i%, "title")
      select case result
         case 1
            msgbox "OK was pressed"
         case 2
            msgbox "Cancel was pressed"
         case 3
            msgbox "Abort was pressed"
         case 4
            msgbox "Retry was pressed"
         case 5
            msgbox "Ignore was pressed"
         case 6
            msgbox "Yes was pressed"
         case 7
            msgbox "No was pressed"
      end select
   next i%

'In this dialog box the second button is the
default.
result=msgbox("Hello World", 3+256, "title")

'In this dialog box the Stop symbol is
displayed.
result=msgbox("Hello World", 3+256+16,
"title")

end sub
```

See Also Dialog Display (Chapter 7)

MsgClose Statement

Description Closes a message window that was opened with the `MsgOpen` statement.

Syntax `MsgClose`

Comments Nothing will happen if there is no open message window.

Example Message Example

See Also Dialog Display (Chapter 7)

MsgOpen Statement

Description Displays a window with a message.

Syntax `MsgOpen msg$,timeout%,isCancel%, isThermometer%[,x%,y%]`

Comments The message can be displayed permanently, or for a specified number of seconds, or until an optional Cancel button is pressed.

The displayed message can be changed by calling the `MsgSetText` statement.

The `timeout%` parameter causes the window to automatically be removed after that number of seconds. The `timeout%` parameter has no effect if its value is zero.

The `isCancel` parameter controls whether or not a Cancel button appears within the window beneath the displayed message. If TRUE, then a Cancel button appears. If not specified, or FALSE, then no Cancel button is created. If a user presses the Cancel button at runtime, a trappable runtime error is generated. In this manner, a message box can be displayed and processing can continue as normal, aborting only when the process is canceled by pressing the Cancel button.

The `isThermometer` parameter controls whether there is a thermometer. If TRUE, then a thermometer is created between the text and the optional Cancel button. The thermometer initially indicated 0% complete, and can be changed using the `MsgSetThermometer` statement.

The optional `x,y` parameters specify the location of the upper left corner of the message box, in twips (1/20th of a point or 1/1440 of an inch). If this point is not specified, then the window is centered on top of the parent.

Only one message window can be opened at any one time. The message window is removed automatically when a script terminates.

Example 1 Message Example

Example 2

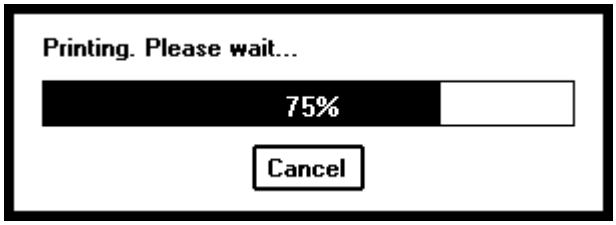
```
MsgOpen "Printing. Please
wait...",0,FALSE,FALSE
```

Example 3

```
MsgOpen "Printing. Please wait...
",0,TRUE,FALSE
```

Example 4

```
MsgOpen "Printing. Please wait...
",0,TRUE,TRUE
MsgSetThermometer 75
```



See Also Dialog Display (Chapter 7)

MsgSetText Statement

Description Changes the text within an open message box (one that was previously opened with `MsgOpen`).

Syntax `MsgSetText newtext$`

Comments The message box is resized to accommodate the new text.

A runtime error will result if a message box is not currently open (using `MsgOpen`).

Example Message Example

See Also Dialog Display (Chapter 7)

MsgSetThermometer Statement

Description Changes the percentage filled in the thermometer of an open message box (one that was previously opened with `MsgOpen`).

Syntax `MsgSetThermometer percentage%`

Comments A runtime error will result if a message box is not currently open (using `MsgOpen`), or if the value of `percentage%` is not between 0 and 100 inclusive.

Example Message Example

See Also Dialog Display (Chapter 7)

Name Statement

Description Renames a file.

Syntax `name oldfile$ as newfile$`

Example

```
sub main()
   'Example of Name statement

   Name "C:\JUNK.TXT" as "C:\NEWJUNK.TXT"
      'renames JUNK.TXT to NEWJUNK.TXT
end sub
```

See Also File Input and Output (Chapter 7)

NetAttach Function

Description Attaches the current user to the specified server using the given user ID and password.

Syntax `NetAttach(server$, user$, password$)`

Comments This function returns the integer TRUE if it is successful in attaching to the specified server or FALSE if the user is already attached, the password verification fails, or the server is not available.

Example

```
sub main()
   'Example of NetAttach

   s$ = AskBox$("Server to attach to:")
   u$ = AskBox$("Username:")
   p$ = AskPassword$("Password:")
   if NetAttach(s$,u$,p$) then
      msgbox("Attach Successful.")
   else
      msgbox("Attach Failed.")
   end if
end sub
```

NDS Note NetAttach functions will return false for NDS users.

See Also Network Functions (Chapter 7)

NetConnectDrive Function

Description Connects a local drive letter with a network path.

Syntax `NetConnectDrive(localpath$, networkpath$ [, [root%], passwd$])`

Comments Returns the integer TRUE if the drive is successfully mapped, otherwise returns FALSE.

If `root%` is specified, then the drive is created with a *false root*. If `passwd$` is specified it is used in attempting to connect the network drive. Note that `root%` and `passwd$` may not be supported in all networks.

The default for `root%` is FALSE, and `passwd$` defaults to none.

The `localpath$` parameter should be specified in the form "E:".

The `networkpath$` parameter is a valid network path, meaning *server/volume:...* or a UNC pathname.

Example

```
sub main()
   'Example of NetConnectDrive

   lp$ = AskBox$("Local drive to connect
to:")
   np$ = AskBox$("Network path to connect:")
   if NetConnectDrive(lp$,np$) then
      msgbox "Drive connected."
   else
      msgbox "Drive connection failed."
   end if
end sub
```

See Also Network Functions (Chapter 7)

NetDetach Function

Description Detaches the current user from the specified file server.

Syntax `NetDetach(server$)`

Comment Returns the integer TRUE if the drive is successfully detached, otherwise returns FALSE.

Caution No safety checks are performed during the detach operation. Detaching from a server is a dangerous thing to do, and this function in no way guarantees that it will be done without causing some sort of system failure.

Example

```
sub main()
   'Example of NetDetach

   s$ = AskBox$("Server To Detach:")
   if NetDetach(s$) then
      msgbox "Server Detached."
   else
      msgbox "Error Detaching Server."
   end if
end sub
```

NDS Note NetDetach functions will return false for NDS users.

See Also Network Functions (Chapter 7)

NetDirectoryRights Function

Description Returns the integer TRUE if the current user has the given rights for the specified network directory path.

Syntax `NetDirectoryRights(path$, rights$)`

Comments The `path$` parameter may be any legal reference to a directory. This includes full paths with drive letter, server/volume, or UNC; or partial paths that are relative to the current directory.

The `rights$` parameter is a series of characters that have network specific meanings. For example, "ROS" on NetWare means Read, Open & Search permissions. If the user has **all** of these permissions for the specified path, the return value is TRUE.

Example

```
sub main()
   'Example of NetDirectoryRights

   p$ = AskBox$("Path to get rights for:")
   r$ = AskBox$("Rights to search for:")
   if NetDirectoryRights(p$,r$) then
      msgbox "You have "+r$+" rights for "+p$
```

```
   else
      msgbox "You do not have "+r$+" rights
for "+p$
   end if
end sub
```

See Also Network Functions (Chapter 7)

NetDisconnectDrive Function

Description Disconnects a local drive letter from a network path.

Syntax `NetDisconnectDrive (networkdrive$)`

Comments The function takes a single string parameter which represents the drive letter that is to be disconnected. It returns an integer value 0 or -1. A 0 (or FALSE) indicates function failure, a -1 (or TRUE) indicates success.

Example

```
sub main()
   'Example of NetDisconnectDrive() function
   dim drive as string
   dim netpath as string

   drive = "k"
   netpath = "sweng_1/sys2:"

   if NetConnectDrive(drive,netpath) then
      msgbox "Drive "+drive+" has been
connected."
      if NetDisconnectDrive(drive) then
         msgbox "Drive "+drive+" has been
disconnected."
      else
         msgbox "Disconnect Fai led."
      end if
   else
      msgbox "Connection Failed."
   end if
end sub
```

See Also Network Functions (Chapter 7)

NetGetDirectoryRights$ Function

Description Returns a string representing the effective rights for the current user on the specified path.

Syntax `NetGetDirectoryRights$(path$)`

Comments The rights are identified as a series of characters compatible with those used by the network operating system that is in use.

Example

```
sub main()
   'Example of NetGetDirectoryRights$()

   p$ = AskBox$("Path to  get rights list
for:")
   r$ = NetGetDirectoryRights$(p$)
   msgbox "Your rights for "+p$+" are "+r$
end sub
```

See Also Network Functions (Chapter 7)

NetMemberOf Function

Description Determines whether the current user is a member of the specified group.

Syntax `NetMemberOf(group$,server$)`

Comments This function returns the integer TRUE or FALSE.

If the `server$` parameter is specified then the groups on that server are searched, otherwise the primary server is used.

If the user USERA is a member of the group SMALLGRP, and SMALLGRP is a member of BIGGROUP, USERA is a member of BIGGROUP.

Example

```
sub main()
   'Example of NetMemberOf

   g$ = AskBox$("Group To Search For:")
   s$ = AskBox$("Server To Search:")
   if NetMemberOf(g$,s$) then
      msgbox "You are a member of that
group."
   else
      msgbox "You are not a member of that
group."
   end if
end sub
```

NDS Note The NetMemberOf function takes two paramers: a server name,and a group name.

Server name: This can be an empty string, or a server name.

If an empty string is passed, and the user is authenticated on an NDS tree, the tree will be searched.

If an empty string is passed, and the user is not authenticated on an NDS tree, the primary bindery server is searched.

If a server name is passed, the server is a not DS server, the server will be searched as a bindery server.

If a server name is passed, and the user is not authenticated on an NDS tree, the server will be searched as a bindery server.

If a server name is passed, the server is a DS server, and the user is authenticated on an NDS tree, the tree will be searched.

Group name: This parameter must be a valid group name.

The type of group name must also match the type of server name passed. If a DS server is passed (and the user is authenticated) but a bindery group name is passed, the results are not valid.

NDS Formats: The group name can be in the following formats, if the server is being searched as an NDS tree:

.CN=GROUPNAME.O=ORG
will be tested for group membership CN=GROUPNAME.O=ORG

.OU=ORGUNIT.O=ORG
will be tested for container membership OU=ORGUNIT.O=ORG O=ORG

O=ORG.NAME.CONTAINER
will be tested for group membership NAME.CONTAINER
(O and OU are used to determine containers)

See Also Network Functions (Chapter 7)

NetStationID Function

Description Returns a network-dependent station id as a string.

Syntax `NetStationID$()`

Comments On Novell networks, the station's Ethernet address is returned.

Example

```
sub main()
   'Example of NetStationID$()

   nsi$ = NetStationID$()
   msgbox "Your ethernet address is : "+nsi$
end sub
```

See Also Network Functions (Chapter 7)

NetUserName Function

Description Returns the user name associated with the given server.

Syntax `NetUserName$([server$])`

Comments If server isn't specified then the primary server is used.

Example

```
sub main()
   'Example of NetUserName

   s$ = AskBox$("Server:")
   nu1$ = NetUserName$()
   nu2$ = NetUserName$(s$)
   msgbox "You are "+nu1$+" on your primary
server, and "+nu2$+" on "+s$+"."
end sub
```

See Also Network Functions (Chapter 7)

NetworkStatus Function

Description Returns an integer representing the network status as a 16 bit (WORD) set of flags.

Syntax `NetworkStatus()`

Comments Each bit of the returned status has different significance. Currently, there are two bit flags.

NS_ACTIVE (0x0001) The network redirector is loaded

NS_LOGGEDON (0x0002) A user is actively logged onto a server/the network.

Normally, this function is compared to the value 3 to determine if additional network calls can or should be made.

Example

```
sub main()
   'Example of NetworkStatus function

   ns% = NetworkStatus()
   msgbox str$(ns%)
   if ns% AND NS_ACTIVE then
      msgbox "Network is active."
   else
      msgbox "Network is not active."
   end if
   if ns% AND NS_LOGGEDON then
      msgbox "Logged On."
   else
      msgbox "Not Logged On."
   end if
end sub
```

See Also Network Functions (Chapter 7)

Not

Description Not operator.

Syntax `NOT expression1`

Returns TRUE if `expression1` is FALSE, otherwise returns TRUE.

If the operand is numeric, then the result is the bitwise NOT of the argument.

Notes If the operand is a floating point value (either single or double), then it is first converted to a long, then a bitwise NOT is performed.

Example

```
sub main()
   'Example of NOT operator
   dim testvar as integer

   testvar = TRUE
   if testvar then
      msgbox "TestVar is TRUE"
   end if

   testvar = FALSE
   if NOT testvar then
      msgbox "TestVar is NOT TRUE"
   end if
end sub
```

See Also Operators (Chapter 7)

Now Function

Description Returns a double-precision number representing the current date and time. The number is returned in days where Dec 20, 1899 is 0.

Syntax `now()`

Example

```
sub main()
   'Example of the Now function
   dim cdt as double

   cdt = Now
   msgbox "Current date is
"+str$(month(cdt))+"/"+str$(day(cdt))+"/"+ ↵
      str$(year(cdt))
end sub
```

See Also Date and Time Functions (Chapter 7)

NS_ACTIVE

Description Constant used by the `NetworkStatus` command.

NS_LOGGEDON

Description Constant used by the `NetworkStatus` command.

Null Function

Description Returns a null string (a string that contains no characters and requires no storage).

Syntax `null[()]`

Comments An empty string (`""`) can also be used to remove all characters from a string. However, empty string still requires some memory for storage. Null strings require no memory.

Example

```
sub main()
   'Example of the Null function
   dim teststr as string

   teststr = "testing"
   msgbox "*"+teststr+"*"
   teststr = null
   msgbox "*"+teststr+"*"
end sub
```

See Also Strings (Chapter 7)

Oct$ Function

Description Returns a string containing the octal equivalent of the specified number.

Syntax `Oct$(number%)`

Comments The returned string contains only the number of octal digits necessary to represent the number.

The `number` parameter can be any type, but is rounded to the nearest whole number before converting to octal.

Example

```
sub main()
   'Example of Oct$() function

   msgbox Oct$(9)
end sub
```

See Also Conversions (Chapter 7)

OKButton Statement

Description Defines an OK button that appears within a dialog box template.

Syntax `OKButton x%,y%,width%,height%`

Comments This statement can only appear within a dialog box template definition (BEGIN DIALOG...END DIALOG).

The `x,y,width,height` parameters are specified in dialog coordinates. The `x,y` position is relative to the upper left corner of the dialog box.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

On Error Statement

Description Defines the action taken when a trappable runtime error occurs.

Syntax `on error {goto <label> | resume next | goto 0}`

Comments The form `on error goto <label>` causes execution to transfer to the specified label when a runtime error occurs.

The form `on error resume next` causes execution to continue to the next line after the line that caused the error.

The form `on error goto 0` causes any existing error trap to be removed.

If an error trap is in effect when the script ends, then an error will be generated.

An error trap is only active within the subroutine or function in which it appears.

Once an error trap has gained control, appropriate action should be taken, and then control should be resumed using the `resume` statement.

If an error occurs within the error handler, the current routines error trap is disabled and a runtime error results.

Example

```
sub main()
   'Example of ON-ERROR statement

   on error goto elabel     'enable error
trap
   error 101                'simulate an
error
   end

elabel:                        'error trap
label
   on error goto 0     'disable error
trapping
   end
end sub
```

See Also Error Trapping (Chapter 7)

Flow Control (Chapter 7)

Open Statement

Description Opens a file.

Syntax

```
open filename$ [for {input | output | append}] as [#]
filenumber%
```

Comments This statement opens a file for a given mode, assigning the open file to the supplied `filenumber`.

The `filename` parameter is a string expression that contains a valid DOS filename.

The `filenumber` parameter is a number between 1 and 255. The `FreeFile()` function can be used to determine an available file number.

The different modes are defined as follows:

Input	Opens an existing file for input.
Output	Opens an existing file, truncating its length to zero, or creates a new file.
Append	Opens an existing file, positioning the file pointer at the end of the file, or creates a new file.

If the `[for mode]` is missing, then `append` is used.

Example Input/Output Example

See Also File Input and Output (Chapter 7)

OpenFileName$ Function

Description Displays the common file open dialog box (from COMMDLG.DLL), allowing the user to select a file.

Syntax `OpenFileName$(title$,extensions$)`

Returns Returns the full DOS pathname of the file the user selected, or an empty string if the user canceled the dialog box.

Comments The `title$` parameter specifies the title that appears on the dialog box's caption.

The `extensions$` parameter specifies the available file types. This string should be in the following format:

`"type:ext[,ext][;type:ext[,ext]]..."`

where *`ext`* is a valid file extension, like *.BAT or *.?F?, and *`type`* is a string that identifies this type to the user.

Example

```
sub main()
   'Example of OpenFileNa me$

   selfile$ = OpenFileName$("Open File","All
Files:*.bmp,  ↵
      *.wmf;Bitmaps:*.bmp; Metafiles:*.wmf")
   msgbox "Selected File = "+selfile$
end sub
```

See Also Dialog Display (Chapter 7)

Option Base Statement

Description Sets the lower bound for array declarations. By default, the lower bound used for all array declarations is 0.

Syntax `Option Base {0 | 1}`

Comments This statement must appear outside of any functions or subroutines.

Example

```
Option Base 1

sub main()
   'Example of O ption Base statement
   dim a(5) as integer

   msgbox str$(lbound(a))
end sub
```

See Also Arrays (Chapter 7)

OptionButton Statement

Description Defines a push button with the specified text that appears within a dialog box template.

Syntax `OptionButton x%,y%,width%,height%,title$`

Comments The `title$` parameter may contain an ampersand character to denote an underlined accelerator, such as "&Font" for `Font`.

This statement can only appear within a dialog box template definition (BEGIN DIALOG...END DIALOG).

The `x%,y%,width%,height%` parameters are specified in dialog coordinates. The `x,y` position is relative to the upper left corner of the dialog box.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

OptionEnabled Function

Description Determines whether an option button is enabled within the current window or dialog box.

Syntax 1 `OptionEnabled(name$)`

Syntax 2 `OptionEnabled(id%)`

Returns Returns the integer TRUE if the specified option button is enabled within the current window or dialog box, otherwise this function returns FALSE.

Comments If an option button is enabled, its value can be set using the `SetOption` statement.

The option button can be referenced given either its `name$` (caption) or its `id%`.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

OptionExists Function

Description Determines whether an option button exists within the current window or dialog box.

Syntax 1 `OptionExists(name$)`

Syntax 2 `OptionExists(id%)`

Returns Returns the integer TRUE if the specified option button exists within the current window or dialog box, otherwise this function returns FALSE.

Comments The option button can be referenced given either its `name$` (caption) or its `id%`.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

OptionGroup Statement

Description Starts a group of option buttons within a dialog box template and defines the name used to determine which option button (from the group of option buttons) is selected when the dialog box ends.

Syntax `OptionGroup .Field`

Comments This statement can only appear within a dialog box template definition (BEGIN DIALOG...END DIALOG).

On exit from the `Dialog` statement, the `.Field` will contain the index of the selected option button, with 0 being the first option button.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

Or

Description Or operator.

Syntax `expression1 OR expression2`

Returns TRUE if either `expression1` is TRUE or `expression2` is TRUE, otherwise FALSE.

If the two operands are numeric, the result is the bitwise OR of the two arguments.

Notes If either of the two operands is a floating point number, the two operands are first converted to longs, then a bitwise OR is performed.

Example

```
sub main()
   'OR statement
   dim a as integer
   dim b as integer

   a = 5
   b = 9
   if (a < 6) OR (b > 8) then
      msgbox "One of the conditions was
true."
   else
      msgbox "Neither condition was true."
   end if

   if (a < 6) OR (b > 9) then
      msgbox "One of the conditions was
true."
   else
      msgbox "Neither condition was tru e."
   end if
end sub
```

See Also Operators (Chapter 7)

PI

Description Pi constant.

Syntax `PI`

Returns 3.14159265358979323846264338327 9

Notes PI can also be determined using the following formula:

4 * atn(1)

Example

```
sub main()
   'Example of the PI function
   dim adouble as double

   adouble = pi
   msgbox str$(adouble)
end sub
```

See Also Math Statements and Functions (Chapter 7)

PO_LANDSCAPE

Description Constant used with the `PrinterSetOrientation` statement to align the paper horizontally.

Returns 2

See Also PrinterGetOrientation

PrinterSetOrientation

PO_PORTRAIT

Description Constant used with the `PrinterSetOrientation` statement to align the paper vertically.

Returns `1`

See Also PrinterGetOrientation

PrinterSetOrientation

PopupMenu Function

Description Creates a popup menu using the string elements in the given array and returns an integer representing the user's response.

Syntax `PopupMenu(MenuItems$())`

Returns Returns the index of the selected item. If no item is selected (the popup menu is canceled), then a value of 1 less than the lower bound is returned.

Comments Each array element is used as a menu item. An empty string results in a separator bar in the menu.

The popup menu is created with the upper left corner at the current mouse position.

A runtime error results if `MenuItems$` is not a single-dimension array.

Only one popup menu can be displayed at a time. An error will result if another script executes this function while a popup menu is visible.

Example

```
sub main()
   'Example of PopupMen u
   dim apps$() as string
   dim result as integer

   AppList apps$
   result = PopupMenu(apps$)
   if result >= lbound(apps$) then
      msgbox "You chose "+apps$(result)
   else
      msgbox "You chose nothing -
"+str$(result)
   end if
end sub
```

See Also Dialog Display (Chapter 7)

Print Statement

Description Writes data to a viewport window.

Syntax `print expression [{, | ;} expression]...`

Comments Strings are written in their literal form, with no enclosing quotes.

Integers and longs are written with an initial space reserved for the sign (space = positive). Additionally, there is a space following each number.

Each `expression` is separated either with a comma (,) or a semicolon (;). A comma means that the next expression is output in the next print zone. A semicolon means that the next expression is output immediately after the current expression. Print zones are defined every 14 spaces.

If the last expression is not followed by a comma or semicolon, then a carriage return is printed to the file. If the last expression in the list ends with a semicolon, no carriage return is printed—the next print statement will output information immediately following the expression. If the last expression in the list ends with a comma, the file pointer is positioned at the start of the next print zone on the current line.

If no viewport window is open, then the statement is ignored. Printing information to a viewport window is a convenient way to output debugging information.

Example

```
sub main()
   'Example of Print

   ViewPortOpen
   ViewPortClear
   print "this is some data"
   sleep 5000
   ViewPortClose
end sub
```

See Also Viewport Window Manipulation (Chapter 7)

Print # Statement

Description Writes data to a disk file.

Syntax `print #filenumber%, expression [{, | ;} expression]...`

Comments The `filenumber%` parameter is a number that is used by DCL to refer to the open file—the number passed to the `open` statement.

Strings are written in their literal form, with no enclosing quotes.

Integers and longs are written with an initial space reserved for the sign (space = positive). Additionally, there is a space following each number.

Each `expression` is separated either with a comma (,) or a semicolon (;). A comma means that the next expression is output in the next print zone. A semicolon means that the next expression is output immediately after the current expression. Print zones are defined every 14 spaces.

If the last expression is not followed by a comma or semicolon, then a carriage return is printed to the file. If the last expression in the list ends with a semicolon, no carriage return is printed - the next print statement will output information immediately following the expression. If the last expression in the list ends with a comma, the file pointer is positioned at the start of the next print zone on the current line.

The `write` statement always outputs information ending with a carriage-return. Thus, if a `print` statement is followed by a `write` statement, the file pointer is positioned on a new line.

The `print` statement can only be used with files that are opened in `output` or `append` modes.

Example

```
sub main()
   open "test.dat" for output as #1
   print #1,10,34,"Hello World";
   a = 10
   s$ = "this is a test"
   print #1,a;s$,
   print #1,67
   close #1
end sub
```

See Also File Input and Output (Chapter 7)

PrinterGetOrientation Function

Description Retrieves the orientation of the default printer—the printer specified in the `device=` line in the `[windows]` section of the WIN.INI file.

Syntax `PrinterGetOrientation()`

Returns Returns the integer PO_PORTRAIT if the printer orientation is set to portrait, otherwise if returns PO_LANDSCAPE.

Comments This function loads the printer driver and therefore may be slow.

Example

```
sub main()
   'Example of PrinterGetOrientation and
   'PrinterSetOrientation
   dim oldorient as integer

   oldorient = PrinterGetOrientation()
   select case oldorient
      case PO_PORTRAIT
         msgbox "Printer is set up for
portrait print."
      case PO_LANDSCAPE
         msgbox "Printer is set up for
landscape print."
   end select
   PrinterSetOrientation  oldorient
end sub
```

See Also Printer Manipulation (Chapter 7)

PrinterSetOrientation Statement

Description Sets the orientation of the default printer—the printer specified in the device= line in the [windows] section of the WIN.INI file.

Syntax PrinterSetOrientation NewSetting%

Comments NewSetting% is PO_PORTRAIT or PO_LANDSCAPE.

This command loads the printer driver for the default printer, and therefore may be slow.

Example

```
sub main()
   'Example of PrinterGetOrientation and
   'PrinterSetOrientat ion
   dim oldorient as integer
   oldorient = PrinterGetOrientation()
   select case oldorient
      case PO_PORTRAIT
         msgbox "Printer is set up for
portrait print."
      case PO_LANDSCAPE
         msgbox "Printer is set up for
landscape print."
   end select
   PrinterSetOrientation oldorient
end sub
```

See Also Printer Manipulation (Chapter 7)

PrintFile Function

Description Invokes the Windows 3.1 shell functions that cause an application to execute and print a file.

Syntax `PrintFile(filename$)`

Returns Returns an integer representing the ID of the executing task.

Comments This function is only available under Windows 3.1.

The application to be executed must be associated with the file extension of the file specified by this command. This association is established in the [Extensions] section of the WIN.INI file. For example, if the Notepad application is associated with the .TXT extension, the Notepad application is started when DCL executes a PrintFile command for a .TXT file.

This command does not support .EXE, .COM, .BAT, or .PIF files.

Example

```
sub main()
   'Example of PrintFile
   taskid = printfile("c:\testfile.txt")
   msgbox "Your file is printing as task
#"+str$(taskid)
end sub
```

See Also Printer Manipulation (Chapter 7)

PushButton Statement

Description Defines a push button within a dialog box template.

Syntax `PushButton x%,y%,width%,height%,title$`

Comments This statement can only appear within a dialog box template definition (BEGIN DIALOG...END DIALOG).

When a push button is selected, the `Dialog` statement ends.

The `x%,y%,width%,height%` parameters are specified in dialog coordinates. The `x,y` position is relative to the upper left corner of the dialog box.

The `title$` parameter may contain an ampersand character to denote an underlined accelerator, such as "&Font" for `Font`.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

QueEmpty Statement

Description Empties the current event queue.

Syntax `QueEmpty`

Comments After this statement, `QueFlush` will do nothing.

Example Queue Example

See Also Keyboard Manipulation (Chapter 7)

Mouse Events (Chapter 7)

QueFlush Statement

Description Plays back events that are stored in the current event queue.

Syntax `QueFlush isSaveState%`

Comments After `QueFlush` is finished, the queue is empty.

The `QueFlush` statement uses the Windows journaling mechanism to replay the mouse and keyboard events stored in the queue. As a result, the mouse position may be changed. Furthermore, events can be played into any Windows application, including DOS applications running in a window.

If `isSaveState` is TRUE, then `QueFlush` saves the state of the CAPSLOCK, NUMLOCK, SCROLL LOCK, and INSERT, and restores the state after the `QueFlush` is complete. If FALSE, these states are not restored.

The function does not return until the entire queue has been played.

Example Queue Example

See Also Keyboard Manipulation (Chapter 7)

Mouse Events (Chapter 7)

QueKeyDn Statement

Description Appends key down events for the specified keys to the end of the current event queue.

Syntax `QueKeyDn Keys$`

Comments The format for `Keys$` is the same as for the `QueKeys KeyString$` parameter, with the exception that parentheses are illegal.

The `QueFlush` command is used to play back the events stored in the current event queue.

Example Queue Example

See Also Keyboard Manipulation (Chapter 7)

QueKeys for list of special keys.

QueKeys Statement

Description Appends keystroke information to the current event queue.

Syntax `QueKeys KeyString$`

Comments To specify any key on the keyboard, simply use that key, such as "a" for lower case a, or "A" for upper case a.

Sequences of keys are specified by appending them together: "abc" or "dir /w".

The keys +, ^, ~, and % are special and must be specified within brackets. For example, to specify the percent, use "{%}".

The keys ()[]{} also have special meaning. To specify one of these keys, enclose it within brackets, such as "{(}".

Keys that are not displayed when you press that key are described within brackets, such as {ENTER} or {UP}.

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}

Keys can be combined with SHIFT, CTRL, and ALT using the reserved keys "+", "^", and "%" respectively:

```
Shift+Enter         "+{ENTER}"
Ctrl+C              "^c"
Alt+F2              "%{F2}"
```

To specify a modifier key combined with a sequence of consecutive keys, group the key sequence within parentheses, as in the following example:

```
Shift+A, Shift+B, Shift+C       "+(abc)"
Ctrl+F1, Ctrl+F2                "^({F1}{F2})"
```

Use "~" as a shortcut for embedding ENTER within a key sequence:

```
"ab~de"
```

To embed quotes, use two quotes in a row:

```
"This is a ""test"" of the system"
```

Key sequences can be repeated using a repeat count within brackets:

```
"{a 10}"     produces 10 "a" keys
"{ENTER 2}      produces 2 Enter keys
```

Example Queue Example

See Also Keyboard Manipulation (Chapter 7)

QueKeyUp Statement

Description Appends key up events for the specified keys to the end of the current event queue.

Syntax `QueKeyUp Keys$`

Comments The format for `Keys$` is the same as for the `QueKeys KeyString$` parameter, with the exception that parentheses are illegal.

The `QueFlush` command is used to play back the events stored in the current event queue.

Example Queue Example

See Also Keyboard Manipulation (Chapter 7)

QueKeys for list of special keys.

QueMouseClick Statement

Description Adds a mouse click to the current event queue.

Syntax `QueMouseClick button%,x%,y%`

Comments A mouse click consists of a mouse button down at position `x,y`, immediately followed by a mouse button up.

The `button` parameter specifies which button to queue: either `VK_LBUTTON` or `VK_RBUTTON`.

The `QueFlush` command is used to play back the events stored in the current event queue.

Example Queue Example

See Also Mouse Events (Chapter 7)

QueMouseDblClk Statement

Description Adds a mouse double click to the current event queue.

Syntax `QueMouseDblClk button%,x%,y%`

Comments A mouse double click consists of a mouse DN/UP/DN/UP at position `x,y`. The events are queued in such a way that a double click is registered during queue playback.

The `button` parameter specifies which button to queue: either `VK_LBUTTON` or `VK_RBUTTON`.

The `QueFlush` command is used to play back the events stored in the current event queue.

Example Queue Example

See Also Mouse Events (Chapter 7)

QueMouseDblDn Statement

Description Adds a mouse double down to the current event queue.

Syntax `QueMouseDblDn button%,x%,y%`

Comments A double down consists of a mouse DN/UP/DN at position `x%,y%`.

The `button%` parameter specifies which button to queue: either `VK_LBUTTON` or `VK_RBUTTON`.

The `QueFlush` command is used to play back the events stored in the current event queue.

Example Queue Example

See Also Mouse Events (Chapter 7)

QueMouseDn Statement

Description Adds a mouse down to the current event queue.

Syntax `QueMouseDn button%,x%,y%`

Comments The `button` parameter specifies which button to queue: either `VK_LBUTTON` or `VK_RBUTTON`.

The `QueFlush` command is used to play back the events stored in the current event queue.

Example Queue Example

See Also Mouse Events (Chapter 7)

QueMouseMove Statement

Description Adds a mouse move to the current event queue.

Syntax `QueMouseMove button%,x%,y%`

Comments The `button` parameter specifies which button to queue: either `VK_LBUTTON` or `VK_RBUTTON`.

The `QueFlush` command is used to play back the events stored in the current event queue.

Example Queue Example

See Also Mouse Events (Chapter 7)

QueMouseUp Statement

Description Adds a mouse up to the current event queue.

Syntax `QueMouseUp button%,x%,y%`

Comments The `button` parameter specifies which button to queue: either `VK_LBUTTON` or `VK_RBUTTON`.

The `QueFlush` command is used to play back the events stored in the current event queue.

Example Queue Example

See Also Mouse Events (Chapter 7)

QueSetRelativeWindow Statement

Description Adjusts all mouse positions relative to the specified window.

Syntax `QueSetRelativeWindow hWnd%`

Comments This statement affects all subsequent `Que...` commands.

The `hWnd%` parameter is a handle to a window in the Windows environment. If `hWnd%` is 0, then the window with the focus is used (i.e., the active window).

The `QueFlush` command is used to play back the events stored in the current event queue.

Example 1 Queue Example

Example 2

```
sub main()
'Adjust mouse coordinates relative to Notepad
hWnd = WinFind("Notepad")
QueSetRelativeWindow hWnd
end sub
```

See Also Mouse Events (Chapter 7)

Queue Example

```
sub main()
   'Example of QUE commands
   dim npWnd as integer
   dim npname as string

   npname = AppFind$("Notepad")
   AppActivate npname
   npWnd = WinFind(npname)
   QueEmpty    'empty the queue
   QueKeyDn "D"
   QueKeyUp "a"
   QueKeys "vid"
   QueMouseClick VK_RBUT TON,1,1
   QueMouseDblClk VK_RBUTTON,1,1
   QueMouseDblDn VK_RBUTTON,1,1
   QueMouseDn VK_RBUTTON,1,1
   QueMouseMove 100,100
   QueMouseUp VK_RBUTTON,100,100
   QueSetRelativeWindow npWnd
   QueFlush TRUE
   QueEmpty
end sub
```

Random Function

Description Generates a random number.

Syntax `Random (min&,max&)`

Returns Returns a long integer greater than or equal to `min` and less than or equal to `max`.

Example

```
sub main()
   'Example of Random function

   ViewPortOpen
   ViewPortClear
   for j% = 1 to 10
      i% = Random(1,100)
      print i%
   next j%
   sleep 5000
   ViewPortClose
end sub
```

See Also Math Statements and Functions (Chapter 7)

Randomize Function

Description Initializes the random number generator with a new seed.

Syntax `Randomize [seed&]`

Comments If `seed` is not specified, then the current value of the system clock is used.

Example

```
sub main()
   'Example of Randomize statement

   Randomize 65
   Randomize
end sub
```

See Also Math Statements and Functions (Chapter 7)

ReadINI$ Function

Description Returns the value of specified item from the specified INI file.

Syntax `ReadINI$(section$,item$[,filename$])`

Comments The `filename$` parameter, if specified, contains the name of the INI file to read. Otherwise, the WIN.INI file is used.

The `section$` parameter specifies the section that contains the desired variable, such as "windows". Section names are specified without the enclosing brackets.

The `item$` parameter specifies which item to retrieve the value of.

Example

```
sub main()
   'Example of ReadIni$

   winshell$ =
ReadIni$("boot","shell","system.ini")
   msgbox winshell$
end sub
```

See Also Environment Statements and Functions (Chapter 7)

ReadINISection Statement

Description Reads all of the item names from a given section of the specified INI file.

Syntax `ReadINISection section$,items$()[,filename$]`

Comments The `filename$` parameter, if specified, contains the name of an INI file. Otherwise, the WIN.INI file is used.

The `section$` parameter specifies the section that contains the desired variables, such as "windows". Section names are specified without the enclosing brackets.

The `items$` parameter refers to the item name on the left side of the = sign. This parameter must be a single dimension array of strings (see Dim statement). On return, this will contain one array element for each variable in the specified INI section. Use the `lbound` and `ubound` functions to determine its size on return.

Example

```
sub main()
   'Example of ReadIniSection
   dim iniitems() as string

   ReadIniSection
"boot",iniitems,"system.ini"
   ViewPortOpen
   print "[Boot] sectio n items from
SYSTEM.INI"
   print " "
   for i% = lbound(iniitems) to
ubound(iniitems)
      print iniitems(i%)
   next i%
   sleep 5000
   ViewPortClose
end sub
```

See Also Arrays (Chapter 7)

Environment Statements and Functions (Chapter 7)

ReDim Statement

Description Redimensions an array, specifying a new upper and lower bound for a given arrays dimensions.

Syntax `ReDim` *`variablename`* `(`*`subscriptRange`*`) [as` *`type`*`],...`

Comments The *`variablename`* parameter specifies the name of an existing array (previously declared using the `dim` statement).

Caution: The `ReDim` statement deletes any data already in the array.

The *`subscriptRange`* parameter specifies the new upper and lower bounds for each dimension of the array using the following syntax:

[*lower%* to] *upper%* [,[*lower%* to] *upper%*]...

If *lower%* is not specified, then 0 is used (or the value set using the option base statement.

Arrays can be dynamically dimensioned by first declaring them with no initial size using the dim statement:

```
dim a$()
```

Then, using the redim statement, the actual size can be specified:

```
redim a$(0 to 8,6 to 10)
```

The number of dimensions of an array cannot be changed once the array has been given dimensions—either by declaring it with initial dimensions using dim or by a previous use of redim.

The *type* parameter can be used to specify the array element type. The following can be used: integer, long, string.

Example

```
sub main()
   'Example of ReDim
   dim stuff(5) as integer
   msgbox str$(ubound(stuff))
   redim stuff(10)
   msgbox str$(ubound(stuff))
end sub
```

See Also Arrays (Chapter 7)

Variables and Constants (Chapter 7)

REM Statement

Description Causes the compiler to skip all characters on that line.

Syntax REM text

Example

```
sub main()
   'Example of REM
REM  this is also a comment
end sub
```

See Also Comments (Chapter 6)

RefreshIni Statement

Description Reloads the WIN.INI file from disk, thus refreshing all WIN.INI settings.

Syntax `RefreshIni`

Comments This forces all hand-edited changes to the WIN.INI file to be activated.

Example

```
sub main()
   'Example of Re freshIni

   RefreshIni
end sub
```

See Also Environment Statements and Functions (Chapter 7)

Reset Statement

Description Closes all open files, writing out all I/O buffers.

Syntax `Reset`

Example

```
sub main()
   'Example of RESET statement

   Reset
      'close all open files (writes out i/o
buffers)
end sub
```

See Also File Input and Output (Chapter 7)

RestoreEnv Function

Description Restores the set of environment variables from the virtual stack for DOS or for Windows saved by the `SaveEnv` function.

Syntax `RestoreEnv([mode$])`

Comments Returns the integer TRUE if the function is successful, otherwise FALSE.

Separate stacks are kept for the Windows and DOS environments.

The `mode$` parameter can be ENV_DOS or ENV_WINDOWS.

If mode is unspecified, the default is ENV_WINDOWS.

Example

```
sub main()
   'Example of SaveEnv and RestoreEnv

   a% = SaveEnv(ENV_WINDOWS)
   a% = RestoreEnv(ENV_WINDOWS)
end sub
```

See Also Environment Statements and Functions (Chapter 7)

Resume Statement

Description Ends an error handler and continues execution.

Syntax `Resume {[0] | next | label}`

Comments The form `resume [0]` causes execution to continue with the statement that caused the error.

The form `resume next` causes execution to continue with the statement following the statement that caused the error.

The form `resume label` causes execution to continue at the specified label.

Example

```
sub main()
   'Example of Resume statement

   on error goto testerror
   error 101
   msgbox "resumed as anticipated"
   on error goto 0
   end

testerror:
   resume next
end sub
```

See Also Error Trapping (Chapter 7)

Return Statement

Description Transfers execution control to the statement following the most recent `gosub`.

Syntax `Return`

Comments A runtime error results if a `return` statement is encountered without a corresponding `gosub` statement.

Example

```
sub main()
   'Examples of GOSUB and RETURN

   for x% = 1 to 5
      gosub mylabel
   next x%
   end

mylabel:
   msgbox "Here we are!"
   return
end sub
```

See Also Flow Control (Chapter 7)

Right$ Function

Description Returns the rightmost `NumChars` characters from a specified string.

Syntax `Right$(str$,NumChars$)`

Comments If `NumChars` is greater than or equal to the length of the string, then the entire string is returned.

If `NumChars` is 0, then an empty string is returned.

Example

```
sub main()
   'Example of right$()
   dim teststr as string

   teststr = "This is a test."
   msgbox teststr
   teststr = right$(teststr,7)
   msgbox teststr
end sub
```

See Also Strings (Chapter 7)

RmDir Statement

Description Removes the specified directory.

Syntax `RmDir dir$`

Comments This command behaves just like the DOS "rd" command.

Example

```
sub main()
   'Example of RmDir

   RmDir "C:\testdir"
end sub
```

See Also File Input and Output (Chapter 7)

Rnd Function

Description Returns a single-precision random number between 0 and 1.

Syntax `Rnd[(number#)]`

Comments If `number#` is omitted, the next random number is returned. Otherwise, the `number#` parameter has the following meaning:

`number# < 0`	Always returns the same number
`number# = 0`	Returns the last number generated
`number# > 0`	Returns the next random number

Example

```
sub main()
   'Example of Rnd function
   ViewPortOpen
   Randomize
   for i% = 1 to 10
      print Rnd(1)
   next i%
   sleep 5000
   ViewPortClose
end sub
```

See Also Math Statements and Functions (Chapter 7)

RTrim$ Function

Description Returns the string with the trailing spaces removed.

Syntax `RTrim$(str$)`

Example

```
sub main()
   'Example of RTrim$()
   dim teststring as string
   teststring = "     testing     "
   msgbox "*" + teststring + "*"
   msgbox "*" + rtrim$(teststring) + "* "
end sub
```

See Also Strings (Chapter 7)

SaveEnv Function

Description Pushes the current environment variable set onto a virtual stack so that they can later be restored by the `RestoreEnv` function.

Syntax `SaveEnv([mode$])`

Comments Returns the integer TRUE if the function is successful, otherwise false.

Separate stacks are kept for the Windows and DOS environments.

The `mode$` parameter can be ENV_DOS or ENV_WINDOWS.

If mode is unspecified, the default is ENV_WINDOWS.

Example

```
sub main()
   'Example of  SaveEnv and RestoreEnv

   a% = SaveEnv(ENV_WINDOWS)
   a% = RestoreEnv(ENV_WINDOWS)
end sub
```

See Also Environment Statements and Functions (Chapter 7)

SaveFileName$ Function

Description Displays the common file save dialog box (from COMMDLG.DLL), allowing the user to select a file. If the file already exists, the user is prompted to overwrite it.

Syntax `SaveFileName$(title$ [,extensions$])`

Returns Returns a full DOS pathname of the file that the user selected.

Comments The `title$` parameter specifies the title that appears on the dialog box's caption.

The `extensions$` parameter specifies the available file types. This string should be in the following format:

"*type*:*ext*[,*ext*][;*type*:*ext*[,*ext*]]..."

where *ext* is a valid file extension, like *.BAT or *.?F?, and *type* is a string that identifies this type to the user. By default, the first extension in appearing within `extensions` is used.

Example

```
sub main()
   'Example of SaveFileName$

   selfile$ = SaveFileName$("Open File",
   "All Files:*.bmp, *.wmf; ↵
      Bitmaps:*.bmp;Metafiles:*.wmf")
   msgbox "Selected File = "+selfile$
end sub
```

See Also Dialog Display (Chapter 7)

Second Function

Description Returns an integer representing the second of the day encoded in the specified `serial#` parameter. The value returned is between 0 and 59 inclusive.

Syntax `second(serial#)`

Comments You can obtain the value for the `serial#` parameter by using the `TimeSerial` or `TimeValue` command.

Example

```
sub main()
   'Example of second() function
   dim dt as double

   dt = Now
   msgbox str$(second(dt))....'current second
end sub
```

See Also Date and Time Functions (Chapter 7)

Seek Statement and Function

Description The `seek` function returns the file pointer for a given file. The `seek` statement sets the file pointer.

Function Syntax `seek(filenumber%)`

Statement Syntax `seek [#] filenumber%,position&`

Comments The `filenumber` parameter is a number that is used by DCL to refer to the open file—the number passed to the `open` statement.

See Also File Input and Output (Chapter 7)

Select...Case Statement

Description Executes a block of DCL statements depending on the value of a given expression.

Syntax

```
select case testexpression
[case expressionlist
  [statement_block]]
[case expressionlist
  [statement_block]]
  :
[case else
  [statement_block]]
end select
```

Comments The `select case` statement uses the following arguments:

testexpression	Any numeric or string expression
statement_block	Any group of DCL statements
expressionlist	Any of the following: *expression* [,*expression*]... *expression* to *expression* is *relational_operator expression*

If the *textexpression* matches any of the expressions contained in *expressionlist*, the accompanying block of DCL statements is executed.

The resultant type of *expression* must be the same as that of *testexpression*.

Multiple expression ranges can be used within a single `case` clause. For example:

```
case 1 to 10,12,15, is > 40
```

Only the *`statement_block`* associated with the first matching expression will be executed.

A `select...end select` expression can also be represented with the `if...then` expression. The use of the `select` statement, however, may be more readable.

Example

```
sub main()
   'Example of SELECT-CASE statement

   i% = 1
   select case i%
      case 1
         msgbox "i% is 1"
      case 2
         msgbox "i% is 2"
   end select
end sub
```

See Also Flow Control (Chapter 7)

SelectBox Function

Description Displays a dialog box containing a listbox of strings. If the user selects an item, the index of that item is returned.

Syntax `SelectBox(title$,prompt$,items$())`

Returns Returns an integer representing the index of the item that the user selected. The first item is 0. The value -1 is returned if the user selects Cancel.

Comments The `items$` parameter must be a single-dimension array of strings, otherwise a runtime error is generated.

The `title$` parameter specifies the name that appears in the dialog box's caption. The `prompt$` parameter specifies the name that appears immediately above the listbox containing the array items.

The dialog box uses the 8 point Helvetica font.

Example

```
sub main()
   'example of SelectBox function
   dim alist() as string
   AppList alist
   result% = SelectBox("Application
List","Pick One",alist)
   msgbox "You selected "+alist(result%)
end sub
```

See Also Dialog Display (Chapter 7)

SelectButton Statement

Description Simulates a mouse click on a button, given the button's name or ID.

Syntax 1 `SelectButton ButtonName$`

Syntax 2 `SelectButton ButtonID%`

Comments You can reference the button by `name$` (caption) or `id%`.

A runtime error is generated if a button with the given `name$` or `id%` cannot be found in the active window.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

SelectComboboxItem Statement

Description Selects an item from a combobox, given the name or ID of the combobox and the name or line number of the item.

Syntax

```
SelectComboboxItem name$ | id%,ItemName$ | ItemNumber%
         [,isDoubleClick%]
```

Comments The combobox can be referenced either by `name$` or by `id%`. The `name$` parameter specifies the text that appears in the static control that immediately precedes the combobox control in the window list (or dialog template).

A runtime error is generated if a combobox with the given `name$` or `id%` cannot be found within the active window, or if an item with the given name or line number cannot be found within that combobox.

If the second parameter is either 0 or an empty string (""), all selections will be removed from the combobox.

The optional `isDoubleClick%` parameter specifies if this item is to be selected via a double click or single click. If not specified, then the item is selected using a single click.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

SelectListboxItem Statement

Description This statement selects an item from a list box, given the name or ID of the listbox and the name or line number of the item.

Syntax
```
SelectListboxItem name$ | id%,ItemName$ | ItemNumber% ↵
        [,isDoubleClick%]
```

Comments The listbox must exist within the current window or dialog, otherwise a runtime error will be generated.

The listbox can be referenced either by name$ or by id%. The name$ parameter specifies the text that appears in the static control that immediately precedes the listbox control in the window list (or dialog template). A runtime error is generated if a listbox with that name cannot be found within the active window.

If the second parameter is 0 or an empty string (""), then all selections are removed from the listbox. For multi-select listboxes, `SelectListBoxItem` will select additional items (i.e., it will not remove the selection from the currently selected items).

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

SendKeys Statement

Description Sends the specified keys to the active application, optionally waiting for the keys to be processed before continuing.

Syntax `SendKeys KeyString$ [,wait%]`

Comments The format for `KeyString$` is the same as that used for `DoKeys`.

If `wait` is TRUE (or not specified), the statement waits for the keys to be completely processed before continuing. Otherwise, execution continues immediately.

Example

```
sub main()
   'Example of SendKeys
   dim alttab as string

   alttab = "%{TAB}"
   msgbox "Press OK to do first Alt-Tab"
   SendKeys alttab,TRUE
   msgbox "Press OK to Alt-Tab back to
original application"
   SendKeys alttab,FALSE
end sub
```

See Also Keyboard Manipulation (Chapter 7)

SetAttr Statement

Description Changes the attributes of the specified file to the given attribute.

Syntax `SetAttr filename$,attribute%`

Comments A runtime error results if the file cannot be found.

The `attribute%` parameter contains the sum of the following attributes.

Constant	Value	Description
ATTR_NORMAL	0	Read-only, archive, subdirectory, and files with no attributes
ATTR_READONLY	1	Read-only files
ATTR_HIDDEN	2	Hidden files
ATTR_SYSTEM	4	System files
ATTR_VOLUME	8	Volume label
ATTR_DIRECTORY	16	MS-DOS directories
ATTR_ARCHIVE	32	Files changed since last backup
ATTR_NONE	64	Files with no attributes

These attributes are the same as those used by DOS.

Example

```
sub main()
   'Example of SetAttr

   SetAttr "C:\AUTOEXEC.BAT",0
      'set file attribute to NORMAL
end sub
```

See Also File Input and Output (Chapter 7)

SetCheckbox Statement

Description Sets the state of the checkbox with the given name or ID.

Syntax 1 `SetCheckbox name$,state%`

Syntax 2 `SetCheckbox id%,state%`

Comments The checkbox can be specified either by its `name$` or using its `id%`. The `name$` parameter is the text of the checkbox label.

If `state` is 1, the box is checked. If `state` is 0, the check is removed. If `state` is 2, then the box is grayed (only applicable for 3-state check boxes).

A runtime error is generated if a check box with the specified name cannot be found in the active window.

This statement has the side-effect of setting the focus to the given checkbox.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

SetEditText Statement

Description Sets the content of an edit control, given its name or ID.

Syntax 1 `SetEditText name$,content$`

Syntax 2 `SetEditText id%,content$`

Comments The `name$` parameter specifies the text that appears within the static control that immediately precedes the edit control in the window list (or dialog template). Alternatively, the `id%` of the edit box can be specified.

This statement has the side-effect of setting the focus to the given edit control.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

SetEnv Function

Description Sets the specified environment variable to the given value for DOS, Windows, or Both.

Syntax `SetEnv(var$, value$[, mode$])`

Comments When you are running Windows, there are two environments. One environment is used for Windows applications and one for DOS applications.

To control which environment you set variables for, use the `mode$` parameter. The `mode$` parameter can be:

-- ENV_DOS, which sets environment variables for DOS applications

-- ENV_WINDOWS, which sets environment variables for Windows.

-- ENV_BOTH, which sets environment variables for both DOS and Windows.

If it is an invalid value, or not set, the default ENV_WINDOWS is used.

Changes made to Windows environment variables are available to all Windows applications—including those that have already been launched.

The function returns TRUE if successful, or FALSE otherwise. If the function fails, the environment may be full.

Example

```
sub main()
   'Example of SetEnv and GetEnv$()
   tmp$ = GetEnv$("TMP",ENV_WINDOWS)
   tv$ = AskBox$("New Value For TMP
Environment Variable:")
   a% = SetEnv("TMP",tv$,ENV_WINDOWS)
   msgbox "New value for TMP is
"+GetEnv$("TMP",ENV_WINDOWS)
   'restore old value
   a% = SetEnv("TMP",tmp$,ENV_WINDOWS)
end sub
```

See Also Environment Statements and Functions (Chapter 7)

SetIcon Statement

Description Specifies which icon to show when `ShowIcon` is called.

Syntax `SetIcon iconfile$ [,icon%]`

Comments The `iconfile$` specifies an .EXE, .DLL, or .ICO file. `Icon%` is the zero-based index into the list of icons the file contains. If unspecified, `icon%` is assumed to be 0.

If the icon cannot be found or is not a valid format, or if the value of `icon%` exceeds the number of icons in the file, a runtime error is generated.

This function has no affect on scripts run from the editor or debugger.

See Also Icons (Chapter 7)

SetIconTitle Statement

Description Sets the title to be displayed under the icon (if shown) for a compiled script.

Syntax `SetIconTitle title$`

Comments This function has no affect on scripts run from the editor or debugger.

See Also Icons (Chapter 7)

SetOption Statement

Description Clicks on the specified option button.

Syntax 1 `SetOption name$`

Syntax 2 `SetOption id%`

Comments The option button can be referenced either by its `name$` (caption) or its `id%`. A runtime error is generated if the option button cannot be found within the active window.

Example Dialog Examples

See Also Dialog Manipulation (Chapter 7)

Sgn Function

Description Returns an integer representing a number is less than, greater than, or equal to zero.

Syntax `sgn(number)`

Comments Returns 1 if `number` is greater than 0.

Returns 0 if `number` is equal to 0.

Returns -1 if `number` is less than 0.

The `number` parameter is a numeric expression of any type.

Example

```
sub main()
   'Example of SGN() function

   i% = -2
   msgbox str$(sgn(i%))
   i% = 0
   msgbox str$(sgn(i%))
   i% = 2
   msgbox str$(sgn(i%))
end sub
```

See Also Math Statements and Functions (Chapter 7)

Shell Function

Description Runs the program (with any specified parameters) contained in `command$`.

Syntax `shell(command$ [,WindowStyle%])`

Returns If successful, the function returns an integer representing the task ID. For other return codes, see "Errors" below.

Comments The optional `WindowStyle%` parameter specifies the state of the application window after execution. It can be any of the following values:

1 Normal window with focus
2 Minimized with focus
3 Maximized with focus
4 Normal window without focus
7 Minimized without focus

An error is generated if unsuccessful. A return value less than or equal to 32 specifies an error.

Errors This function returns the value 31 if there is no association for the specified file type or if there is no association for the specified action within the file type. The other possible error values are as follows:

0 System was out of memory, executable file was corrupt, or relocations were invalid.
2 File was not found.
3 Path was not found.
5 Attempt was made to dynamically link to a task, or there was a sharing or network-protection error.
6 Library required separate data segments for each task.
8 There was insufficient memory to start the application.
10 Windows version was incorrect.
11 Executable file was invalid. Either it was not a Windows application or there was an error in the .EXE image.
12 Application was designed for a different operating system.
13 Application was designed for MS-DOS 4.0.
14 Type of executable file was unknown.
15 Attempt was made to load a real-mode application (developed for an earlier version of Windows).
16 Attempt was made to load a second instance of an executable file containing multiple data segments that were not marked read-only.
19 Attempt was made to load a compressed executable file. The file must be decompressed before it can be loaded.
20 Dynamic-link library (DLL) file was invalid. One of the DLLs required to run this application was corrupt.
21 Application requires Microsoft Windows 32-bit extensions.

Example

```
sub main()
   'Example of Shell function

   taskid% = Shell("notepad.exe",1)
end sub
```

See Also Flow Control (Chapter 7)

ShowIcon Statement

Description Displays an icon on the desktop while the script is running.

Syntax `ShowIcon [show%]`

Comments Normally, an icon does not show up on the desktop while a compiled script is running.

If `show%` is set to TRUE (the default value), an icon will appear on the desktop when the script is run. The purpose of the icon is to allow the user to stop the script if the user has been permitted this authority (through the `EnableStopScript` command).

This function has no affect on scripts run from the editor or debugger.

See Also Icons (Chapter 7)

Sin Function

Description Returns a double-precision number representing the sine of a given angle.

Syntax `sin(angle#)`

Comments The `angle` parameter is given in radians.

Example

```
sub main()
   'Example of Sin() function
      result# = sin(0)
   msgbox str$(result#)
   result# = sin(1)
   msgbox str$(result#)
end sub
```

See Also Math Statements and Functions (Chapter 7)

Sleep Statement

Description Causes the script to pause for a specified number of milliseconds.

Syntax `sleep milliseconds&`

Comments While paused, other applications can execute.

Example

```
sub main()
   'Example of Sleep command

   msgbox "Press OK to sleep 5 seconds"
   sleep 5000
end sub
```

See Also Flow Control (Chapter 7)

SleepUntil Function

Description Pauses the script until the specified time is reached.

Syntax

```
SleepUntil(time$[,showdialog%[,text$[,caption$[,cancel%  ↵
        [,minimizebox]]]], usenettime%)
```

Comments The integer TRUE is returned if the pause is completed normally. FALSE is returned if an error is encountered or if the user cancels the operation.

The `time$` value must be in one of the following formats:

12 Hour - "HH:MM xm"

24 Hour - "HH:MM"

HH = hours (may be 1 or 2 digits for 12 hour format, but must be 2 digits for 24 hour)

MM = minutes

xm = am/pm indicator (the 'x' will be replaced by 'a' or 'p')

Colons (":") are expected between the hour and minute portions of the time. In the 12-hour format, a space is expected before the "xm" portion.

Any deviation from this time format will generate a runtime error during execution.

You can set `showdialog%` to TRUE or FALSE. If TRUE, a dialog is displayed with the specified features. The `text$` parameter specifies the text to display in the dialog box. The dialog resizes to display the text. If text not specified, no text is used. The `caption$` parameter specifies the caption to be used. If not given, then "Sleeping Until"+time$ is used.

If `cancel%` is specified, a Cancel button appears in the dialog box. Clicking on the Cancel button stops the sleep and returns FALSE.

If `minimizebox%` is specified, then the dialog can be minimized. It will show a clock icon.

Example

```
Set usenettime% to TRUE to use network time
instead of DOS time.
sub main()
   'Example of SleepUntil

   t$ = time$()       'get current time
   t$ = left$(t$,5)   'get hours and
                     'minutes only
   m$ = right$(t$,2)  'get minutes value
   h$ = left$(t$,2)   'get hour value
   x = val(m$)     'get the value of the
minutes
   y = val(h$)     'get value of hour string
   x = x + 2          'increment minutes
                     'to wait 2 minutes
if x > 59 then
      y = y + 1
      x = x - 59
   end if
   h$ = str$(y)       'convert hours
                     'back to string
   h$ = right$(h$,len(h$)-1)
      'get rid of leading blank from
conversion
   if len(h$) = 1 then h$ = "0" + h$
   m$ = str$(x)       'convert minutes
                       'back to string
likewise
```

```
      m$ = right$(m$,len(m$)-1)
   if len(m$) = 1 then m$ = "0" + m$
   t$ = h$+":"+m$     'reconstruct time
                      'value
   retval% = SleepUntil(t$,TRUE,  ↵
      "Press CANCEL to quit.",  ↵
      "Sleeping Until "+t$,TRUE,TRUE)
end sub
```

See Also Flow Control (Chapter 7)

Snapshot Statement

Description Takes a snapshot of a particular section of the screen and saves it to the clipboard.

Syntax `snapshot [spec%]`

Comments The `spec` parameter specifies the screen area as follows:

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 window of the active window

Before the snapshot is taken, each application is updated. This ensures that any application in the middle of drawing will have a chance to finish before the snapshot is taken.

There is a slight delay if the specified window is large.

Example

```
sub main()
   'Example of SnapShot statement

   SnapShot 4  'active window
   taskid% = Shell("pbrush.exe")
   DoKeys "+{INSERT}"    'paste clipboard
                         'into
paintbrush
end sub
```

See Also Clipboard Manipulation (Chapter 7)

Space$ Function

Description Returns a string containing the specified number of spaces.

Syntax `space$(NumSpaces%)`

Comments NumSpaces must be between 0 and 32767.

Example

```
sub main()
   'Example of Space$() function

   stuff$ = null
   msgbox "*"+stuff$+"*"
   stuff$ = space$(10)
   msgbox "*"+stuff$+"*"
end sub
```

See Also Strings (Chapter 7)

Sqr Function

Description Returns a double-precision number representing the square root of a given value.

Syntax `sqr(number#)`

Comments The `number` parameter must be greater than or equal to 0.

Example

```
sub main()
   'Example of SQR function

   msgbox str$(sqr(100))
end sub
```

See Also Math Statements and Functions (Chapter 7)

Stop Statement

Description Stops execution of a script.

Syntax `stop`

Comments This command terminates execution of the current script and displays the message: "Stopped at line X", where X is the line number containing the stop statement. All open files are closed. All open DDE channels are closed.

Example

```
sub main()
   'Example of Stop statement
   Stop
end sub
```

See Also Flow Control (Chapter 7)

Str$ Function

Description Returns a string representation of the given number.

Syntax 1 `str$(number%)`

Syntax 2 `str$(number&)`

Syntax 3 `str$(number!)`

Syntax 4 `str$(number#)`

Comments If `number` is negative, then the returned string will contain a leading minus sign.

If `number` is positive, then the returned string will contain a leading space. Singles are printed using only 7 significant digits. Doubles are printed using 15-16 significant digits.

Example

```
sub main()
   'Example of str$() function
   i% = 3
   s$ = str$(i%)
   msgbox s$
end sub
```

See Also Conversions (Chapter 7)

Strings (Chapter 7)

StrComp Function

Description Compares two strings.

Syntax `StrComp(string1$,string2$ [,compare%])`

Returns Returns an integer value indicating the result of comparing the two string arguments:

`0` `string1$ = string2`

`1` `string1$ > string2$`

`-1` `string1$ < string2$`

Comments The `StrComp` function compares two strings and returns an integer indicating the result of the comparison. The comparison can be either case-sensitive or case-insensitive, depending on the value of the optional `compare%` parameter:

`0` Case-sensitive comparison
`1` Case-insensitive comparison

If `compare%` is not specified, then the comparison is case-sensitive (`0`).

Example

```
sub main()
   'Example of StrComp function

   'Case-sensitive comparison
   teststr$ = "This is a test string"
   result% = StrComp(teststr$,"THIS IS A TEST
STRING",0)

   select case result%
      case -1
         msgbox "teststr$ is less than the
compare string"
      case 0
         msgbox "teststr$ is equal to the
compare string"
      case 1
         msgbox "teststr$ is greater than
compare string"
   end select
```

```
   'Case-insensitive comparison
   result% = StrComp(teststr$,"THIS IS A TEST
STRING",1)
   select case result%
      case -1
         msgbox "teststr$ is less than the
compare string"
      case 0
         msgbox "teststr$ is equal to the
compare string"
      case 1
         msgbox "teststr$ is greater than
compare string"
   end select
end sub
```

See Also Strings (Chapter 7)

String$ Function

Description Returns a string of `number%` length consisting of a repetition of the specified filler character.

Syntax 1 `string$(number%,CharCode%)`

Syntax 2 `string$(number%,str$)`

Comments If `CharCode` is specified, the character with this ASCII value is used as the filler character.

If `str` is specified, then the first character of this string is used as the filler character.

Example

```
sub main()
   'Example of String$() function

   s$ = String$(10,65)
   msgbox s$
   s$ = String$(10,"B")
   msgbox s$
end sub
```

See Also Strings (Chapter 7)

StringSort Function

Description Sorts a one-dimensional array of strings in ascending order.

Syntax
```
StringSort list$()
```

Comments If an array of more than one dimension is specified, a runtime error is generated.

See Also Strings (Chapter 7)

Sub...End Sub Statement

Description Declares a subroutine.

Syntax
```
sub name[(parameter [as type]...)]
end sub
```

Comments Parameters are passed to a subroutine by reference, meaning that any modification to a passed parameter changes that variable in the caller. To avoid this, simply enclose variable names in parentheses, as in the following example function calls:

```
UserSub 10,12,(j)
```

If a subroutine is not to receive a parameter by reference, the optional `byval` keyword can be used:

```
sub Test byval FileName as string
end sub
```

A subroutine terminates when one of the following statements is encountered:

```
end sub
exit sub
```

The name of the subroutine must follow DCL naming conventions. It cannot include type declaration characters.

Subroutines can be recursive.

See Also Procedure Statements (Chapter 7)

SystemFreeMemory Function

Description Returns a long integer representing the number of bytes of free memory.

Syntax `SystemFreeMemory()`

Example

```
sub main()
   'Example of SystemFreeMemory

   msgbox "Free Memory
="+str$(SystemFreeMemory)
end sub
```

See Also Environment Statements and Functions (Chapter 7)

SystemFreeResources Function

Description Returns an integer representing the percentage of free system resources.

Syntax `SystemFreeResources()`

Comments The returned value is between 0 and 100.

Example

```
sub main()
   'Example of SystemFreeResources

   msgbox "Free Resources
="+str$(SystemFreeResources)+"%"
end sub
```

See Also Environment Statements and Functions (Chapter 7)

SystemMouseTrails Statement

Description Turns on or off mouse trails.

Syntax `SystemMouseTrails state%`

Comments The setting is saved in the INI file permanently.

This option is only available under Windows 3.1.

Example

```
sub main()
   'Example of SystemMouseTrails

   SystemMouseTrails TRUE
   msgbox "Move the Mouse Around Now"
   SystemMouseTrails FALSE
end sub
```

See Also Environment Statements and Functions (Chapter 7)

SystemRestart Statement

Description Restarts Windows, much like the Window Setup program.

Syntax `SystemRestart`

Example

```
sub main()
   'Example of SystemRestart
   'WARNING:   this will restart Windows if
you run it

   SystemRestart
end sub
```

See Also Environment Statements and Functions (Chapter 7)

SystemTotalMemory Function

Description Returns a long integer representing the total available free memory in Windows in bytes.

Syntax `SystemTotalMemory()`

Example

```
sub main()
   'Example of SystemTotalMemory

   msgbox "Total System Memory =" +  ↵
      str$(SystemTotalMemory)
end sub
```

See Also Environment Statements and Functions (Chapter 7)

SystemWindowsDirectory$ Function

Description Returns the directory where Windows is stored, such as "C:\WINDOWS".

Syntax `SystemWindowsDirectory$()`

Example

```
sub main()
   'Example of SystemWindowsDirectory$

   msgbox SystemWindowsDirectory$
end sub
```

See Also Environment Statements and Functions (Chapter 7)

SystemWindowsVersion$ Function

Description Returns the Windows version, such as "3.0" or "3.1".

Syntax `SystemWindowsVersion$()`

Example

```
sub main()
   'Example of SystemWindowsVersion$

   msgbox SystemWindowsVersion$
end sub
```

See Also Environment Statements and Functions (Chapter 7)

Tan Function

Description Returns a double-precision number representing the tangent of the specified angle.

Syntax `tan(angle#)`

Comments The `angle` parameter is given in radians.

Example

```
sub main()
   'Example of Tan() function

   i# = Tan(1)
   msgbox str$(i#)
end sub
```

See Also Math Statements and Functions (Chapter 7)

Text Statement

Description Defines a text control in a dialog template.

Syntax `Text x%,y%,width%,height%,title$`

Comments The purpose of `Text` controls is simply to display text.

The `title$` parameter will be truncated if the width of the control is insufficient to hold the entire content. The `title$ parameter` may contain an ampersand character to denote an underlined accelerator, such as "&Font" for `Font`.

This statement can only appear within a dialog box template definition (BEGIN DIALOG...END DIALOG).

The `x,y,width,height` parameters are specified in dialog coordinates. The `x,y` position is relative to the upper left corner of the dialog box.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

TextBox Statement

Description Defines a text-entry field that appears within a dialog box template.

Syntax `TextBox x%,y%,width%,height%,.Field`

Comments This statement can only appear within a dialog box template definition (BEGIN DIALOG...END DIALOG).

The `x,y,width,height` parameters are specified in dialog coordinates. The `x,y` position is relative to the upper left corner of the dialog box.

When the dialog box is created, the content of the `.Field` is used to set the initial content of the textbox. When the `Dialog` statement returns, the `.Field` is used to determine the final content of the textbox. The `.Field` always contains a string.

Example Dialog Examples

See Also Dialog Creation (Chapter 7)

Time$ Statement and Function

Description The `time$` assignment statement sets the system time; the `time$` function returns the system time.

Assignment Syntax `time$ = newtime$`

Comments This statement sets the system time to the time contained in the specified string.

The format for `newtime$` must be one of the following:

`HH`

`HH:MM`

`HH:MM:SS`

A 24 hour clock is used.

Example

```
sub main()
   'Example of Time command

   time$ = "21:30:40"
end sub
```

Function Syntax `time$()`

Returns Returns the system time as an 8 character string.

Comments The format of the returned string is `HH:MM:SS`.

Example

```
sub main()
   'Example of Time$() function

   msgbox Time$()
end sub
```

See Also Date and Time Functions (Chapter 7)

Timer Function

Description Returns a long integer representing the number of seconds that have occurred since midnight.

Syntax `timer`

Example

```
sub main()
   'Example of Timer function
   dim tval as long

   tval = Timer
   msgbox str$(tval)
end sub
```

See Also Date and Time Functions (Chapter 7)

TimeSerial Function

Description Returns a double-precision number representing the given time with today's date. The number is returned in days where Dec 30, 1899 is 0.

Syntax `TimeSerial(hour%,minute%,second%)`

Example

```
sub main()
   'Example of TimeSerial function
   dim tser as double

   tser = TimeSerial(21,40,44)
end sub
```

See Also Date and Time Functions (Chapter 7)

TimeValue Function

Description Returns a double-precision number representing the time contained in the specified string argument.

Syntax `TimeValue(time_string$)`

Comments This function interprets the passed `time_string$` parameter looking for a valid time specification. Time specifications vary depending on the international settings contained in the INTL section of the WIN.INI file.

The `time_string$` parameter can contain valid time items separated by time separators such as colon (:) or period (.). The time items must follow the ordering determined by the current time format settings in use by Windows.

Time strings can contain an optional date specification, but this is not used in the formation of the returned value.

If a particular time item is missing, then the missing time items are set to zero. For example, the string "10 pm" would be interpreted as "22:00:00".

Example

```
sub main()
   'Example of TimeValue function
   dim tval as double

   tval = TimeValue("21:40:44")
end sub
```

See Also Date and Time Functions (Chapter 7)

Trim$ Function

Description Removes leading and trailing spaces.

Syntax `trim$(str$)`

Returns Returns a copy of the passed string expression (`str$`) with leading and trailing spaces removed.

Example

```
sub main()
   'Example of Trim$()
   dim teststring as string
```

```
    teststring = "     testing     "
    msgbox "*" + teststring + "*"
    msgbox "*" + trim$(teststring) + "*"
end sub
```

See Also Strings (Chapter 7)

TRUE

Description Constant.

Returns -1

Comments Used in conditionals and boolean expressions.

TYPE_DOS

Description Constant.

Returns `1`

Comment Used with the `AppType` function to indicate a DOS application.

TYPE_WINDOWS

Description Constant.

Syntax `2`

Comment Used with the `AppType` function to indicate a Windows application.

UBound Function

Description Returns an integer representing the upper bound of the specified dimension of the specified array variable.

Syntax `ubound(ArrayVariable() [,dimension%])`

Comments The first dimension (1) is assumed if `dimension` is not specified.

Example

```
sub main()
   'Examples of UBOUND
   dim ia1(8) as integer
   dim ia2(65 to 70) as integer

   msgbox str$(ubound(ia1))
   msgbox str$(ubound(ia2))
end sub
```

See Also Arrays (Chapter 7)

UCase$ Function

Description Returns the uppercase equivalent of the specified string.

Syntax `ucase$(str$)`

Example

```
sub main()
   'Example of UCase$
   dim teststr as string

   teststr = "this is a test"
   msgbox teststr
   teststr = ucase$(teststr)
   msgbox teststr
end sub
```

See Also Strings (Chapter 7)

Val Function

Description Converts a given string expression to a double-precision number.

Syntax `val(number$)`

Comments The `number$` parameter can contain any of the following:

- Leading minus sign (for non hex or octal numbers only)
- Hexadecimal number in the format: &H<hex digits>
- Octal number in the format: &O<octal digits>
- Floating point number, which can contain a decimal point and optional exponent

Spaces, tabs, and linefeeds are ignored.

If `number$` does not contain a number, 0 is returned.

The `val()` function continues to read characters from the string up to the first non-numeric character.

The `val()` function always returns a double-precision floating point value. This value is forced to the data type of the assigned variable.

Example 1

```
sub main()
   'Example of Val function

   teststr$ = "123.3"
   tval# = val(teststr$)
   msgbox str$(tval)
end sub
```

Example 2 The following table shows valid strings and their numeric equivalent:

"1 2 3"	123
"12.3"	12.3
"&HFFFF"	-1
"&O77"	64
" 12.345E-02"	.12345

See Also Conversions (Chapter 7)

Strings (Chapter 7)

ViewportClear Statement

Description Clears the open viewport window.

Syntax `ViewportClear`

Comments The statement has no effect if no viewport is opened.

Example

```
sub main()
   'Example of ViewPort commands

   'Create a veiwport window
   ViewPortOpen "My ViewPort"
   For i% = 1 to 20
      Print i%
   Next i%
   msgbox "Press OK to clear the viewport."
   ViewPortClear
   msgbox "Press OK to close the viewport."
   ViewPortClose
end sub
```

See Also Viewport Window Manipulation (Chapter 7)

ViewportClose Statement

Description Closes an open viewport window.

Syntax `ViewportClose`

Example

```
sub main()
   'Example of ViewPort commands

   'Create a veiwport window
   ViewPortOpen "My ViewPort"
   For i% = 1 to 20
      Print i%
   Next i%
   msgbox "Press OK to clear the viewport."
   ViewPortClear
   msgbox "Press OK to close the viewport."
   ViewPortClose
end sub
```

See Also Viewport Window Manipulation (Chapter 7)

ViewportOpen Statement

Description Opens a viewport window.

Syntax `ViewportOpen [title$ [,x%,y% [,width%,height%]]]`

Comments The optional `title$` parameter specifies the text to appear in the viewport's caption. The `x` and `y` parameters specify an optional initial position in twips, and the optional `width` and `height` parameters specify an optional initial width and height for the viewport window.

This statement has no effect if a viewport window is already open.

Combined with the `print` statement, a viewport window is a convenient place to output debugging information.

The viewport window is closed when the DCL host application is terminated.

The buffer size for the viewport is 32K. Information from the start of the buffer us removed to make room for additional information being appended to the end of the buffer.

The following keys work within a viewport window:

Up	Scroll up by one line
Down	Scroll down by one line
Home	Scroll to the first line in the viewport window
End	Scroll to the last line in the viewport window
PgUp	Scroll the viewport window down by one page
PgUp	Scroll the viewport window up by one page
Ctrl+PgUp	Scroll the viewport window left by one page
Ctrl+PgDn	Scroll the viewport window right by one page

Only 1 viewport window can be open at any one time. Any scripts with `print` statements will output information into the same viewport window.

Example

```
sub main()
   'Example of ViewPort commands

   'Create a veiwport window
   ViewPortOpen "My ViewPort"
   For i% = 1 to 20
      Print i%
   Next i%
   msgbox "Press OK to clear the viewport."
   ViewPortClear
   msgbox "Press OK to close the viewport."
   ViewPortClose
end sub
```

See Also Viewport Window Manipulation (Chapter 7)

VK_LBUTTON

Description Constant used with the `QueMouse...` commands to represent the left button.

Value 1

See Also Mouse Events

VK_RBUTTON

Description Constant used with the `QueMouse...` commands to represent the right button.

Value 2

See Also Mouse Events

VLine Statement

Description Scrolls the window with the focus up or down by the specified number of lines.

Syntax `VLine [lines%]`

Comments If the `lines` parameter is omitted, then the window is scrolled down by 1 line.

Example

```
sub main()
   'Examples of VLINE
   ViewPortOpen
   ViewPortClear
   for i% = 1 to 50
      Print "Here is so me test data."
   next i%
   VLine 50
   sleep 2000
   VLine -50
   ViewPortClose
end sub
```

See Also Window Manipulation (Chapter 7)

VPage Statement

Description Scrolls the window with the focus up or down by the specified number of pages.

Syntax `VPage [pages%]`

Comments If the `pages` parameter is omitted, then the window is scrolled down by 1 page.

Example

```
sub main()
   'Examples of VPage

   ViewPortOpen
   ViewPortClear
   for i% = 1 to 50
      Print i%
   next i%
   VPage 1
   sleep 2000
   VPage -1
   ViewPortClose
end sub
```

See Also Window Manipulation (Chapter 7)

VScroll Statement

Description Sets the thumb mark on the vertical scroll bar attached to the current window.

Syntax `VScroll percentage%`

Comments The position is given as a percentage of the total range associated with that scroll bar. For example, if the `percentage%` parameter is 50, then the thumb is positioned in the middle of the scroll bar.

Example

```
sub main()
   'Example of VSCROLL

   ViewPortOpen
   ViewPortClear
```

```
   for i% = 1 to 50
      Print "Test data for  viewport scroll
test."
   next i%
   sleep 2000
   VScroll 50  '50 percent scroll
   sleep 2000
   VScroll 1      'no scroll
   sleep 2000
   ViewPortClose
end sub
```

See Also Window Manipulation (Chapter 7)

WaitForTaskCompletion Function

Description Waits until the task specified by `taskid%` is exited.

Syntax `WaitForTaskCompletion taskid%`

Comments The `taskid%` is the return value from the `Shell` statement.

If `taskid%` is not currently running, control returns immediately.

Example

```
sub main()
   'Example of WaitForTaskCompl etion
   taskid% = Shell("notepad",1)
   'The next statement pauses until Notepad
   'is shut down
   WaitForTaskCompletion taskid%
   msgbox "All done."
end sub
```

See Also Flow Control (Chapter 7)

Weekday Function

Description Returns an integer representing the day of the week of the date encoded in the specified `serial` parameter. The value returned is between 1 and 7 inclusive where 1 is Sunday.

Syntax `weekday(serial#)`

Comments You can obtain the value for the `serial#` parameter by using the `DateSerial` or `DateValue` command.

Example

```
sub main()
   'Example of Weekday()

   wday% = weekday(Now)
   select case wday%
      case 1
         msgbox "Today Is Sunday"
      case 2
         msgbox "Today Is Monday"
      case 3
         msgbox "Today Is Tuesday"
      case 4
         msgbox "Today Is Wednesday"
      case 5
         msgbox "Today Is Thursday"
      case 6
         msgbox "Today Is Friday"
      case 7
         msgbox "Today Is Saturday"
   end select
end sub
```

See Also Date and Time Functions (Chapter 7)

While...Wend Statement

Description Repeats a statement or group of statements while a condition is TRUE.

Syntax
```
while <condition>
  [<statement>]
wend
```

Example
```
sub main()
   'Example of WHILE-WEND

   i% = 4
   while i% > 1
      msgbox "not yet"
      i% = i% -1
   wend
   msgbox str$(i%)
end sub
```

See Also Flow Control (Chapter 7)

WinActivate Statement

Description Activates the window with the given name or window handle.

Syntax 1
```
WinActivate window_name$
```

Syntax 2
```
WinActivate hWnd%
```

Comments The `window_name$` parameter specifies the name that appears on the desired application's title bar. The parameter is not case-sensitive. Optionally, a partial name can be used, such as "Word" for "Microsoft Word".

A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example:

```
WinActivate "Notepad|Find"
```

In the above example, the top level windows are first searched for a name that includes the string "Notepad". When found, the windows owned by the found window are searched for one that contains the string "Find" in its window title.

If the `hWnd%` parameter is specified rather than the `window_name$` parameter, then focus is set immediately to the window with that handle.

Windows without captions cannot be activated using this command.

Example

```
sub main()
   'Example of WinActivate

   appn$ = AppFind$("Notepad")
   WinActivate appn$
end sub
```

See Also Window Manipulation (Chapter 7)

WinClose Statement

Description Closes the given window.

Syntax `WinClose [window_name$]`

Comments If no `window_name$` parameter is specified, the window with the focus is closed.

The `window_name$` parameter specifies the name that appears on the desired application's title bar. The parameter is not case-sensitive. Optionally, a partial name can be used, such as "Word" for "Microsoft Word".

A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example:

```
WinActivate "Notepad|Find"
```

In the above example, the top level windows are first searched for a name that includes the string "Notepad". When found, the windows owned by the found window are searched for one that contains the string "Find" in its window title.

This command differs from the `AppClose` command in that this command operates on the current window rather than the current top-level window. The current window can be an MDI child window, a popup window, or a top-level window.

Example

```
sub main()
   'Example of WinClose

   appn$ = AppFind$("Notepad")
   WinClose appn$
end sub
```

See Also Window Manipulation (Chapter 7)

WinFind Function

Description Returns an integer representing a handle to the specified window.

Syntax `WinFind(name$)`

Comments The `name$` is specified using the same format as that used by the `WinActivate` statement.

Example

```
sub main()
   'Example of WinFind

   appn$ = AppFind$("Notepad")
   hWnd% = WinFind(appn$)
   msgbox str$(hWnd%)
end sub
```

See Also Window Manipulation (Chapter 7)

WinList Function

Description Fills the passed array with the handles to all the top level windows.

Syntax `WinList hWnd$()`

Comments After calling this function, use the `lbound()` and `ubound()` functions to determine the new size of the array.

Example

```
sub main()
   'Example of WinList
   dim hWindows() as integer
   WinList hWindows
   for i% = lbound(hWindows) to
ubound(hWindows)
      msgbox str$(hWindows(i%))
   next i%
end sub
```

See Also Window Manipulation (Chapter 7)

WinMaximize Statement

Description This command maximizes the specified window.

Syntax `WinMaximize [window_name$]`

Comments If no `window_name$` parameter is specified, the window with the focus is maximized.

The `window_name$` parameter specifies the name that appears on the desired application's title bar. The parameter is not case-sensitive. Optionally, a partial name can be used, such as "Word" for "Microsoft Word".

A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example:

```
WinActivate "Notepad|Find"
```

In the above example, the top level windows are first searched for a name that includes the string "Notepad". When found, the windows owned by the found window are searched for one that contains the string "Find" in its window title.

This command differs from the `AppMaximize` command in that this command operates on the current window rather than the current top-level window. The current window can be an MDI child window, a popup window, or a top-level window.

Example

```
sub main()
   'Example of WinMaximize
   appn$ = AppFind$("Notepad")
   WinMaximize appn$
end sub
```

See Also Window Manipulation (Chapter 7)

WinMinimize Statement

Description Minimizes the specified window.

Syntax `WinMinimize [window_name$]`

Comments If no `window_name$` parameter is specified, the window with the focus is minimized.

The `window_name$` parameter specifies the name that appears on the desired application's title bar. The parameter is not case-sensitive. Optionally, a partial name can be used, such as "Word" for "Microsoft Word".

A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example:

```
WinActivate "Notepad|Find"
```

In the above example, the top level windows are first searched for a name that includes the string "Notepad". When found, the windows owned by the found window are searched for one that contains the string "Find" in its window title.

This command differs from the `AppMinimize` command in that this command operates on the current window rather than the current top-level window. The current window can be an MDI child window, a popup window, or a top-level window.

Example

```
sub main()
   'Example of WinMinimize

   appn$ = AppFind$("Notepad")
   WinMinimize appn$
end sub
```

See Also Window Manipulation (Chapter 7)

WinMove Statement

Description Moves the specified window.

Syntax `WinMove x%,y% [window_name$]`

Comments This command moves the given window to the given x,y position. If no `window_name$` parameter is specified, then the window with the focus is moved.

The `window_name$` parameter specifies the name that appears on the desired application's title bar. The parameter is not case-sensitive. Optionally, a partial name can be used, such as "Word" for "Microsoft Word".

A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example:

```
WinActivate "Notepad|Find"
```

In the above example, the top level windows are first searched for a name that includes the string "Notepad". When found, the windows owned by the found window are searched for one that contains the string "Find" in its window title.

This command differs from the `AppMove` command in that this command operates on the current window rather than the current top-level window. The current window can be an MDI child window, a popup window, or a top-level window. When moving child windows, remember that the `x%` and `y%` parameters are relative to the client area of the parent window.

Example

```
sub main()
   'Example of WinMove

   appn$ = AppFind$("Notepad")
   for i% = 0 to 100
      WinMove i%, i%, appn$
   next i%
end sub
```

See Also Window Manipulation (Chapter 7)

WinRestore Statement

Description Restores the specified window.

Syntax `WinRestore [window_name$]`

Comments If no `window_name$` parameter is specified, the window with the focus is restored.

The `window_name$` parameter specifies the name that appears on the desired application's title bar. The parameter is not case-sensitive. Optionally, a partial name can be used, such as "Word" for "Microsoft Word".

A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example:

```
WinActivate "Notepad|Find"
```

In the above example, the top level windows are first searched for a name that includes the string "Notepad". When found, the windows owned by the found window are searched for one that contains the string "Find" in its window title.

This command differs from the `AppRestore` command in that this command operates on the current window rather than the current top-level window. The current window can be an MDI child window, a popup window, or a top-level window.

Example

```
sub main()
   'Example of WinRestore

   appn$ = AppFind$("Notepad")
   WinRestore appn$
end sub
```

See Also Window Manipulation (Chapter 7)

WinSize Statement

Description Resizes the specified window.

Syntax `WinSize width%,height% [window_name$]`

Comments This command resizes the given window to the specified width and height. If no `window_name$` parameter is specified, the window with the focus is resized.

The window_name$ parameter specifies the name that appears on the desired application's title bar. The parameter is not case-sensitive. Optionally, a partial name can be used, such as "Word" for "Microsoft Word".

A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example:

```
WinActivate "Notepad|Find"
```

In the above example, the top level windows are first searched for a name that includes the string "Notepad". When found, the windows owned by the found window are searched for one that contains the string "Find" in its window title.

This command differs from the `AppSize` command in that this command operates on the current window rather than the current top-level window. The current window can be an MDI child window, a popup window, or a top-level window.

Example

```
sub main()
   'Example of WinSize
   appn$ = AppFind$("Notepad")
   for i% = 1 to 200
      WinSize i%, i%, appn$
   next i%
end sub
```

See Also Window Manipulation (Chapter 7)

Word$ Function

Description Extracts words from a text source.

Syntax `word$(text$,first%[,last%])`

Returns Returns a single word or sequence of words between `first` and `last`.

Comments The `first` parameter specifies the first word in the sequence to return. If `last` is not specified, then only that word is returned. If `last` is specified, then all words between `first` and `last` will be returned, including all spaces, tabs, and end-of-lines that occur between those words.

Word are separated by spaces, tabs, and end-of-lines.

If `first` is greater than the number of words in `text$`, then an empty string is returned.

If `last` is greater than the number of words in `text$`, then all words from `first` to the end of text are returned.

Example

```
sub main()
   'Example of WORD$() and WORDCOUNT
functions
   dim teststr as string
   teststr = "The quick brown fox jumps over
the lazy dog."
   for i% = 1 to wordcount(teststr)
      msgbox word$(teststr,i%,i%)
   next I%
end sub
```

See Also Strings (Chapter 7)

WordCount Function

Description Returns an integer representing the number of words in the specified text.

Syntax `WordCount(text$)`

Comments Word are separated by spaces, tabs, and end-of-lines.

Example

```
sub main()
   'Example of WORD$() and WORDCOUNT
functions
   dim teststr as string
   teststr = "The quick brown fox jumps over
the lazy dog."
   for i% = 1 to wordcount(teststr)
      msgbox word$(teststr,i%,i%)
   next i%
end sub
```

See Also Strings (Chapter 7)

Write # Statement

Description Writes a list of expressions to the specified file.

Syntax `write [#]filenumber% [,expressionlist]`

Comments The file referenced by `filenumber` must be opened in either `output` or `append` mode.

The `filenumber` parameter is a number that is used by DCL to refer to the open file—the number passed to the `open` statement.

See Also File Input and Output (Chapter 7)

WriteINI Statement

Description Writes a new value into an INI file.

Syntax `WriteINI section$,ItemName$,value$[,filename$]`

Comments The `filename$` parameter, if specified, contains the name of an INI file. If `filename$` is not specified, the WIN.INI file is used.

The `section$` parameter specifies the section which contains the desired variables, such as "windows". Section names are specified without the enclosing brackets.

The `ItemName$` parameter specifies which item from within the given section you want to change. If `ItemName$` is an empty string (""), then the entire section specified by `section$` is deleted.

The `value$` parameter specifies the new value for the given item. If `value$` is an empty string (""), then the item specified by `ItemName$` is deleted from the INI file.

Example

```
sub main()
   'Example of WriteIni statement

   WriteIni
"anewsection","anewitem","value","win.ini"
end sub
```

See Also Environment Statements and Functions (Chapter 7)

WS_MAXIMIZED

Description Constant used with the `AppSetState` and `AppGetState` statements to indicate a maximized window state.

Value `1`

WS_MINIMIZED

Description Constant used with the `AppSetState` and `AppGetState` statements to indicate a minimized window state.

Value `2`

WS_RESTORED

Description Constant used with the `AppSetState` and `AppGetState` statements to indicate a normal window state.

Value `3`

Xor

Description Xor operator.

Syntax `expression1 XOR expression2`

Returns If both operands are relational, then `XOR` returns the logical exclusive OR of `expression1` and `expression2`. In other words, `XOR` returns TRUE only if both operands are not equal.

If both operands are numeric, the result is the bitwise XOR of the arguments.

Notes If either of the two operands is a floating point number, the two operands are first converted to longs, then a bitwise XOR is performed.

Example

```
sub main()
   'Example of XOR
   dim a as integer
   dim b as integer
   a = 5
   b = 9
   if (a < 6) XOR (b > 8) then
      msgbox "Both conditions were not the
same."
   else
      msgbox "Both conditions were the same--
either ↵
        TRUE or FALSE."
   end if
   if (a < 6) XOR (b > 9) then
      msgbox "Both conditions were not the
same."
   else
      msgbox "Both conditions were the same--
either ↵
        TRUE or FALSE."
   end if
end sub
```

See Also Operators (Chapter 7)

Year Function

Description Returns an integer representing the year of the date encoded in the specified `serial` parameter. The value returned is between 100 and 9999 inclusive.

Syntax `year(serial#)`

Comments You can obtain the value for the `serial#` parameter by using the `DateSerial` or `DateValue` command.

Example

```
sub main()
   'Example of Year
   msgbox str$(year(Now))    'display current
year
end sub
```

See Also Date and Time Functions (Chapter 7)

Notes

Index

%

%, 21

&

&, 21

.

.Field, 56

A

Array$, 56
Arrays, 26, 81

B

Breakpoints, 69

C

Cancel button, 56
Check box, 57
Combo box, 58
Comments, 18, 83
Compiler, 78
Constants, 21, 84
    user-defined, 22
Context-sensitive help, 48

D

Data types, 19, 81
DCL Editor
    closing file, 34
    context-sensitive help, 48
    creating new script, 32
    editing text, 38
    exiting, 32
    finding text, 39
    help, 48
    macro recorder, 42
    making and distributing executables, 35
    opening existing file, 33
    printing files, 37
    replacing text, 40
    running scripts, 41
    saving scripts, 34
    shortcut keys, 49
    starting, 31
    Status Bar, 33
    syntax checking, 42
    Toolbar, 32
    windows, 48
Debugger
    breakpoints, 69
    context-sensitive help, 75
    editing dialogs, 74
    editing text, 71
    exiting, 75
    finding text, 72
    help, 74
    replacing text, 73
    running a script, 68
    shortcut keys, 75
    starting, 42, 67
    toolbar, 68
    tracing, 69
    watch variables, 69
Dialog box
    capturing, 59
    creating or modifying, 53
    sample script, 63
    saving new, 61
    testing, 59
Dialog box controls
    adding, 54
    clearing, 59
    cutting, copying, pasting, 59
    defining, 55
    duplicating, 59
Dialog box file
    opening existing, 60

saving under new name, 61
Dialog Editor
context-sensitive help, 62
exiting, 65
help, 62
running standalone, 62
shortcut keys, 65
starting, 42, 51
Status Bar, 54
Toolbar, 53
Do...Loop statement, 30

E

Editing text, 38
Error handling, 80
Executable
distributing, 36
making, 35
Exiting
DCL Editor, 32
Debugger, 75
Dialog Editor, 65
Expression evaluation, 79

F

File
closing, 34
opening existing, 33
printing, 37
recent, 34
saving as executable, 35
saving existing, 35
Finding text, 39
For...Next statement, 29
Functions, 24, 80
passing parameters, 25
user-defined, 24

G

General characteristics of DCL, 77

H

Help, getting, 48

I

Icons, arranging, 48
If...Then...Else statement, 27
Integers, 21

K

Keys, text editing, 49

L

Limitations, 85
List box, 57

M

Macro, recording, 42

O

OK button, 56
Operator precedence, 83
Operators, 22
Option group and buttons, 57
Overview of DCL, 15

P

Printing
a script, 37
setting up for, 38
Push button, 56

R

Recent files, 34
Recorded events
application focus switch, 44
dialog box interaction, 47
keyboard activity, 45
menu commands, 47
mouse activity, 45
window management, 46
window scrolling, 44
Recorder
events captured by. see Recorded Events
macro, 42
Reference, passing parameters by, 25
Release notes, 16
Replacing text, 40
Running script, 41

S

Saving a file, 35

Saving a script, new, 34
Script
    checking syntax, 42
    creating new, 32
    execution, 78
    printing, 37
    recording, 42
    running, 41
    saving new, 34
Select Case statement, 28
Shortcut keys
    DCL Editor, 49
    Debugger, 75
    Dialog Editor, 65
Starting
    DCL Editor, 31
    Debugger, 67
    Dialog Editor, 51
Status Bar
    DCL Editor, 33
    Dialog Editor, 54
String
    finding, 39
    replacing, 40
Strings, 19
Structures, 78
Subroutines, 23, 80
Syntax, checking, 42
System requirements, 16

T

Text
    editing, 38
    editing keys, 49
    finding, 39
    replacing, 40
Text box, 58
Text$, 56
Toolbar
    DCL Editor, 32
    Debugger, 68
    Dialog Editor, 53
Tracing, 69

V

Value, passing parameters by, 25
Variables, 19, 78

W

Watch pane, opening, 70
Watch variable
    deleting, 71
While...Wend statement, 30
Windows, working with, 48