COREL®

WordPerfect® OFFICE X5

User Guide for PerfectScript™

Contents

Introduction

Welcome to the *Corel® WordPerfect® Office User Guide for PerfectScript™*!

PerfectScript™ is a command-based macro-programming language that you can use to automate tasks in Corel® WordPerfect®, Corel® Quattro Pro®, and Corel® Presentations™. This documentation contains basic information about creating simple PerfectScript macros, as well as detailed, technical information about creating more complex PerfectScript macros.

This documentation contains the following sections:

- "Understanding macro concepts" on page 3 explains the concepts that are associated with macros, and shows how these concepts apply to PerfectScript macros
- "Getting started with macros" on page 83 introduces you to the PerfectScript utility, which you can use to create macros quickly and easily
- "Creating macros" on page 89 examines how to create macros, either by migrating ones that already exist or by recording or writing new ones
- "Creating dialog boxes for macros" on page 101 describes how to use a dialog box to create an interface between the application and the user
- "Debugging macros" on page 135 demonstrates how to find and correct any errors in your macros

This documentation also contains a glossary.

Please see the PerfectScript Help file (**psh.chm**) for the following additional sections:

- "PerfectScript Command Reference" documents the syntax elements and macro commands for PerfectScript
- "WordPerfect Command Reference" documents the system variables and macro commands for WordPerfect
- "Quattro Pro Command Reference" documents the syntax elements and macro commands for Quattro Pro, for both PerfectScript and the native Quattro Pro macro-programming language
- "Presentations Command Reference" documents the macro commands for Presentations
- "Gallery of sample macros" provides sample macros for PerfectScript and WordPerfect

Understanding macro concepts

When performing repetitive or complex tasks in WordPerfect Office, you can save time by using PerfectScript macros. In this section, you'll learn the concepts that are associated with macros, and you'll learn how these concepts apply to PerfectScript macros.

This section contains the following topics:

- Understanding macros
- Using expressions in macro statements
- Using command statements in macros
- Using assignment statements in macros
- Using conditional statements in macros
- Using loop statements in macros
- Using calling statements in macros
- Using comment statements in macros
- Accessing external applications in macros
- Learning more about macros

Understanding macros

A *macro* specifies a sequence of actions that you can quickly repeat later. For example, a macro can automate a WordPerfect task such as setting the margins, selecting a font, or creating a merge file.

To create macros for WordPerfect Office, you can use the *PerfectScript* macro-programming language. PerfectScript is called a "command-based language" because it uses *macro commands* to store the results of an action rather than storing the individual steps that are used to carry out that action.

You can also create macros for Quattro Pro by using the native macro-programming language for the application. For information, please see "Understanding the native Quattro Pro macro language" in the Quattro Pro Command Reference section of the PerfectScript Help file (**psh.chm**).

You can also use Microsoft® Visual Basic® for Applications (VBA) to create macros for WordPerfect Office. For detailed information about VBA and VBA macros, please see the *Corel WordPerfect Office User Guide for VBA* (**vba_ug.pdf**).

A macro consists of a set of instructions or *statements*. By using the various types of macro statements, you can create PerfectScript macros that automate anything from a basic task to a complex procedure. For more information about macro statements, see "Understanding macro statements" on page 4.

Through the use of macro statements, PerfectScript lets you create macros that access applications outside of WordPerfect Office. For more information, see "Accessing external applications in macros" on page 77.

For even more information about macros, you can consult additional resources for WordPerfect Office. For information, see "Learning more about macros" on page 81.

The proper form of macro components is governed by a set of rules, or *syntax*. For more information about macro syntax, see "Understanding macro syntax" on page 5.

If you structure your macros well, they will function well — and be much easier to edit. For more information about macro structure, see "Understanding macro structure" on page 5.

Understanding macro statements

If a macro represents a set of instructions, then a macro statement represents a single step in those instructions. The simplest macro consists of only one statement, while the most complex macro consists of multiple statements that are performed in sequence.

A group of related statements is called a "statement block."

Some statements require an *expression*, which is a formula that represents a value. For more information about expressions, see "Using expressions in macro statements" on page 8.

By combining expressions with other macro components, you can create any of the following types of statements:

- *command statements* — consist of a macro command, which represents a single instruction (typically, an action). For more information, see "Using command statements in macros" on page 47.

- *assignment statements* — assign a value to an expression. For more information, see "Using assignment statements in macros" on page 54.
- *conditional statements* — execute a statement (or a group of statements) when a specified condition is met. For more information, see "Using conditional statements in macros" on page 54.
- *loop statements* — execute a statement (or a group of statements) a specified number of times until (or while) an expression is true. For more information, see "Using loop statements in macros" on page 57.
- *calling statements* — call a statement (or a group of statements). For more information, see "Using calling statements in macros" on page 59.
- *comment statements* — contain notes that explain the purpose of a macro without affecting its play. For more information, see "Using comment statements in macros" on page 77.

Understanding macro syntax

The proper form of macro components is governed by a set of rules, or syntax. For a macro to work properly, its code must use the correct syntax — that is, the code must be "syntactically correct."

For each macro component that is described in this documentation, details on proper macro syntax are included.

Some macro statements are too lengthy to fit into a single line of macro code. If your macro editor automatically inserts a hard return at the end of every line, you must insert an underscore character (_) at the end of each line that wraps. For information on specifying a macro editor, see "To specify settings for editing macros" on page 86.

Understanding macro structure

If you structure your macros well, they will function well — and be much easier to edit.

You can structure a macro in several ways. The basic function of a macro is to accomplish a task by following a series of steps, so the ideal structure for a macro depends on the task involved — and on the amount of code that is required to carry out that task. For example, if a macro involves multiple tasks that require large amounts of code, you can make the macro more manageable by breaking it into smaller pieces (called *subroutines* — see “Understanding subroutines” on page 59).

From a structural standpoint, the two main types of macros are as follows:

- *sequential macros* — progress in steps from start to finish. For more information, see “Understanding sequential macros” on page 6.
- *procedural macros* — progress in steps based on user intervention. For more information, see “Understanding procedural macros” on page 7.

Understanding sequential macros

A sequential macro progresses in steps from start to finish. All steps are taken in the required order, and the code is written to suit that purpose.

An example of a sequential macro follows:

```
HardReturn ()
HardReturn ()
GetString( var1; "Enter Name"; "Data Entry"; 100 )
Type (Text: var1)
HardReturn ()
GetString( var2; "Enter Address"; "Data Entry"; 100 )
Type( var2 )
HardReturn ()
HardReturn ()
Type (Text: "Dear " + var1 + ":")
HardReturn ()
HardReturn ()
Type (Text: "Yaddah Yaddah Yaddah")
HardReturn ()
HardReturn ()
HardReturn ()
Type (Text: "Sincerely,")
HardReturn ()
HardReturn ()
HardReturn ()
HardReturn ()
```

```
Type (Text: "Paul McRussell")
HardReturn ()
Type (Text: "Manager, Eat-a-Chicken Burger, Anywhere, USA")
```

Understanding procedural macros

A procedural macro progresses in steps based on user intervention, through the use of functions and procedures (see “Understanding functions and procedures” on page 61). Using functions and procedures in a macro lets the programmer compartmentalize code so it can be called from anywhere in the macro. Compartmentalization breaks logical pieces of code into smaller segments, and these segments can be separated by use of the `Label`, `Function`, and `Procedure` commands (see “Understanding subroutines” on page 59). Smaller pieces of code are easier to work with, and they are also easier to debug.

An example of a procedural macro follows:

```
HardReturn ()
HardReturn ()
//Call the function to get the name
sName = GetName()
Type (Text: sName)
HardReturn ()
//Call the function to get the address
sAddress = GetAddress()
Type (sAddress )
HardReturn ()
HardReturn ()
Type (Text: "Dear " + sName + ":")
HardReturn ()
HardReturn ()
Type (Text: "Yaddah Yaddah Yaddah")
HardReturn ()
HardReturn ()
HardReturn ()
Type (Text: "Sincerely,")
HardReturn ()
HardReturn ()
HardReturn ()
HardReturn ()
Type (Text: "Paul Russell")
HardReturn ()
Type (Text: "Manager, Eat-a-Burger, Anywhere, USA")
Function GetName()
GetString( sName; "Type in the name of the addressee"; _
"Enter Name"; 100 )
RETURN( sName )
EndFunction
Function GetAddress()
GetString( sAddress; "Type in the address of the addressee"; _
"Enter Address"; 100 )
RETURN( sAddress )
EndFunction
```

Using expressions in macro statements

Macros consist of statements. Some macro statements involve an action that must be captured as an expression. An expression is a formula that represents a value.

To create expressions, you use the following macro components:

- *variables* — store a single value at a time, but this value can change during macro play. For more information, see "Understanding variables" on page 10.
- *constants* — store a single value at a time, and this value cannot change during macro play. For more information, see "Understanding constants" on page 25.
- *operators* — are symbols (such as `+`, `-`, `*`, and `%`) that combine variables and constants to determine a value. For more information, see "Understanding operators" on page 25.

Understanding expressions

Expressions are created by combining variables or constants (or both) with operators — or by combining other expressions with operators.

The following examples contain expressions that involve variables and operators.

Example	Result
`x := "John Doe"`	The variable `x` equals the character string `John Doe`.
`vLeftMargin := 5i`	The variable `vLeftMargin` equals the measurement `5i`.
`ResultOfOperation := 3 + 4`	The variable `ResultOfOperation` equals `7` (that is, the result of the numeric expression `3 + 4`).
`z := z + 1`	The variable `z` equals the value of `z + 1`. However, a variable can contain only one value at a time, so the original value of `z` is lost unless previously assigned to another variable.
`x := y > 1`	The variable `x` equals the result of the relational expression `y > 1` (that is, `x` equals `True` if `y` contains a value greater than `1`, or it equals `False` if `y` contains a value less than or equal to `1`).

`If (y>1)`
`Beep`
`EndIf`
	The result of `y>1` is evaluated without assigning the result to a variable. The computer beeps if the value of `y` is greater than `1` (that is, if the result of the expression `y>1` equals `True`). The beep is skipped if the value of `y` is less than or equal to `1` (that is, if the result of the expression equals `False`).

The following example contains expressions that involve variables, constants, and operators. The value `vCount` is used as a variable, while the values `0`, `4`, and `-1` are used as constants. The operators `-` and `=` are used to create expressions from these values: `vCount - 1`, `vCount = 0`, and `vCount = 4`.

```
Function BeepBeep(vCount)
Repeat
Beep
Wait(3)
vCount := vCount - 1
```

```
Until(vCount = 0)
Return
EndFunc
ForEach(vCount; {1; 2; 3; 4; 5})
If(vCount = 4)
Break
EndIf
BeepBeep(vCount)
Wait(5)
EndFor

MessageBox(x; "BREAK"; "Variable vCount equals 4"; IconInformation!)

Quit
```

For more information about the types of expressions that you can create, see “Understanding expression types” on page 40.

Understanding variables

A variable stores a single value at a time, but this value can change during macro play.

Variables must be “declared” before they can be used. Declaring a variable instructs PerfectScript to set aside memory for the variable.

Assigning a value to — or “initializing” — a variable involves pointing that variable to the memory cell where its desired value is stored. If desired, variables can be initialized with a value at the time of declaration. Although the value of a variable can belong to any data type, the most common data types for variables are numbers and character strings.

For more information about declaring and initializing variables, see “Declaring and initializing variables” on page 12.

Unlike other programming languages, PerfectScript does not force the programmer to specify the type of data to be stored in a variable.

When a variable is declared, it is assigned to one of four types:

- *local variables* — pertain only to the current macro. By default, variables are automatically declared local if no variable type is specified. For more information, see "Working with local variables" on page 13.
- *global variables* — pertain to the current macro and to macros that are called by the `Run` and `Chain` commands. For more information, see "Working with global variables" on page 15.
- *persistent variables* — pertain to any PerfectScript macro, for as long as PerfectScript is running. For more information, see "Working with persistent variables" on page 16.
- *constant variables* — represent a value that cannot change during macro play. For more information, see "Working with constant variables" on page 18.

Two additional kinds of variables require special attention:

- A *system variable* is a type of macro command that contains current system information such as the current chart type or the default directory. For example, the PerfectScript system variable `ErrorNumber` contains the error value of a `Cancel`, `Error`, or `Not Found` condition (as illustrated in line 44 of the annotated macro sample ASSERT.WCM in the PerfectScript Help file [**psh.chm**]). Similarly, the WordPerfect system variable `?PathMacros` assigns the path and name of the default folder for WordPerfect macros to a variable named `vMacroPath`, which is updated to reflect any changes to the directory. For more information about system variables, see "Understanding macro commands" on page 47.
- An *implicit variable* is a variable that is defined by PerfectScript. For example the `MacroDialogResult` variable contains the control value of the button that releases a dialog box (see "Releasing dialog boxes by using PerfectScript code" on page 132).

The type of a variable determines its visibility (or *scope*) and its duration in memory, so it's important to understand when to use each variable type. If you try to access a variable from a line of code in which that variable is not visible, an "out-of-scope" error is generated.

In addition to variable type, the following factors determine the scope of a variable:

- where the variable is declared — for example, in the main body, in a function, in a procedure, in another macro, or in a separate program altogether
- which line of code is currently executing

You can check whether a variable exists. For more information, see “Determining whether variables exist” on page 18.

When a variable is no longer required, you can discard it. For more information, see “Discarding variables” on page 19.

If you want to assign a collection of data to a single variable name, you can use an *array.* The rules for using arrays are the same as for using variables. For more information, see “Working with arrays” on page 20.

Declaring and initializing variables

When you declare a macro, you specify a name for it.

For best results, it is highly recommended that you give your variables a descriptive name. Variable names have the following standards:

- They must begin with a letter.
- They can include any other combination of letters or numbers.
- They must be 50 characters or fewer in length.
- They are not case-sensitive.

Optionally, you can initialize a variable at its time of declaration by using an assignment operator (`:=` or `=`) to specify a value.

By following a few simple conventions for naming variables, you can make your macro code easier to understand. For instance, variables that have a string value should have a name that begins with a lowercase `s`, as in the following examples:

```
sFirstName := "Dave"

sAddress := "1625 East Nowhere St."

sBirthday := "6/12/69"
```

Similarly, variables that have a numeric value should have a name that begins with a lowercase `n`, as in the following examples:

```
nAge := 25

nTotal := 145.97
```

In the preceding examples, all declared variables are local, by default, because no variable type is specified.

Working with local variables

Local variables pertain only to the current macro. Local variables are the default variable type and, as such, should be used in most situations. You can use the PerfectScript programming commands `Declare` or `Local` to create local variables.

Variables that are declared in user-defined functions and user-defined procedures are local to those subroutines. For more information about subroutines, functions, and procedures, see “Understanding subroutines” on page 59.

Local variables can be declared in the following way:

```
Declare ( sQReport )
```

or

```
Local ( sQReport )
```

Local variables can be declared and intitialized in the following way:

```
sQReport := "Q4"

Declare ( sQReport := "Q4" )
```

or

```
sQReport := "Q4"

Local ( sQReport := "Q4" )
```

If you want, you can use the `Declare` command or the `Local` command to declare and initialize more than one local variable at a time. Variables are separated by a semicolon (`;`), as in the following example:

```
Declare (sFilename := "c:\test.wpd"; sTemp; nCount; cMainCount:=10)
```

When a local variable is declared, it is assigned to the local-variable table. Variables in the local-variable table are visible only until the end of the level of code in which they are declared. The level of code usually refers to the main body or a subroutine (that is, a function or procedure). Consider the following sample code:

```
FileNew

/* Declare sName as a local variable and initialize to the string
value "Dave" */

vName := "Dave"
```

```
/* Call the procedure */

TypeName ( )

Quit

Procedure TypeName ( )

/* This variable is out of scope. It has not been declared in the
procedure TypeName */

Type ( vName )

HardReturn

EndProcedure
```

The preceding code assigns the string `"Dave"` to the variable. It then calls the procedure which tries to type the contents of the variable. Because the variable is out of scope within this procedure, the following error occurs when playing the macro:

```
Undefined variable ‘VNAME’ has been referenced. Check line 9 of macro
file ‘test.wcm.’
```

Consider the following sample code:

```
...

/* Variables declared in the main body are visible in the main body */

vNameMain := "Dave"

NewScope ( )

/* When processing this Procedure vNameMain is not visible */

NewScope2 ( )

/* When processing this Function vNameMain is not visible */

...

Quit

// Subroutines . . .

Procedure NewScope ( )

/* Local variables declared in a procedure are visible only in that
procedure */

NameNewScope := "Fred"

...

EndProcedure

Function NewScope2 ( )
```

```
/* Local variables declared in a function are visible only in that
function */
vNameNewScope2 := "John"
...
EndFunction
```

In the preceding code, all three variables (`VNameMain`, `NameNewScope`, and `VNameNewScope2`) are named differently. However, these variables could have been named the same and still have been completely unique variables — each one holding different data — because they are each declared at a different level of the macro and therefore each have their own scope.

Working with global variables

Global variables pertain to the current macro and to macros that are called by the `Run` and `Chain` commands. Although a necessity in some cases, global variables should be used with care. You can use the PerfectScript programming command `Global` to create global variables.

Global variables can be declared in the following way:

```
Global ( nCount )
```

Global variables can be declared and initialized in the following way:

```
Global ( sFilename := "c:\Expense.wpd" )
```

If you create two variables with the same name (for example, `Declare x` and `Global x`), the following statement specifies that the global variable `x` is assigned the value `5`:

```
Global x:=5
```

If you want, you can use the `Global` command to declare and initialize more than global variable at a time by separating variables with a semicolon (`;`).

When a variable is declared global, it is assigned to the global-variable table. Variables assigned to the global-variable table are in scope anytime after they are declared, and they exist until the end of the macro in which they are declared. If a global variable is declared on the very first line of a macro, it is accessible in the main body, in subroutines, and in other macros that are started with the commands `Run` or `Chain`. The following is an example of a global variable in use:

```
...
/* The global variable is not yet declared and not yet accessible */
```

```
Global ( sGlobalName := "Fred" )

/* Any reference after this to the variable sGlobalName accesses the
global variable */

...

/* Call the function */

DoSomething ( )

Type ( sGlobalName )

...

/* sGlobalName ceases to exist when the macro ends */

Quit

Procedure DoSomething ( )

/* Change the value of the global variable to "Dave" */

sGlobalName := "Dave"

...

EndProcedure
```

In the preceding example, the procedure is called after the global variable `sGlobalName` is declared and initialized. Inside this procedure, the contents of the variable are changed from `"Fred"` to `"Dave"`. The commands `DoSomething` and `Procedure DoSomething` allow you to start another macro from within the current macro and, therefore, to access and change any variables that are declared global in the current macro.

Working with persistent variables

Persistent variables pertain to any PerfectScript macro, for as long as PerfectScript is running. Although a necessity in some cases, persistent variables should be used with care. You can use the PerfectScript programming command `Persist` to create persistent variables — in much the same way as you can use the `Global` command to create global variables.

Persistent variables can be declared in the following way:

```
Persist ( VariableName )
```

Persistent variables can be declared and initialized in the following way:

```
Persist ( VariableName := Value )

PersistAll ( On! )

...
```

```
/* All variables declared in the default manner are now persistent
instead of local*/

VariableName := Value

...

PersistAll ( Off! )
```

The preceding example uses the `PersistAll` command to change the default variable-declaration method from local to persistent and back again. All variables between `PersistAll ( On! )` and `PersistAll ( Off! )` are declared as `Persistent` variables. This technique is useful when you want an entire block of variables to be persistent.

If you want, you can use the `Persist` command to declare and initialize more than persistent variable at a time by separating variables with a semicolon (`;`).

When a variable is declared persistent, it is assigned to the persistent-variable table. Variables in the persistent table remain in scope and exist until PerfectScript shuts down.

PerfectScript does not shut down until all the applications that use PerfectScript (WordPerfect, Quattro Pro, and Presentations) have shut down.

Persistent variables are visible during merges and, as such, provide an effective method for passing values between macros and merges. If you need to use data during a merge, use persistent variables. For best results, give persistent variables a descriptive name, and denote their data type.

The following example of a persistent variable requires the use of two macros and includes a test that determines whether the variable has been initialized.

The first macro in this example is as follows:

```
Persist ( sAppName := "WordPerfect Suite 8" )

MessageBox( retVal; sAppName; "Left margin equals: " + ?MarginLeft )

Run ( "Macro2.wcm" )
```

The second macro in this example is as follows:

```
MessageBox(retVal; sAppName; "Right margin equals: " + ?MarginRight)
```

The following example illustrates scope by using local and persistent variables:

```
Persist ( x := "This is persistent variable x" )

CreateOutline()
```

```
// Original variable value remains unchanged.
MessageBox ( retVal ; "Information"; "The variable x = " + x )
Quit
Procedure CreateOutline ( )
/* PerfectScript will look first at the local variable table. If a
variable exists in that table, that variable will be used before the
persistent variable. By creating a local variable inside the
function, we will force PerfectScript to find the local variable.
This local variable x will be destroyed when execution returns from
this subroutine. */
Local ( x := 0 )
ForNext ( x; 1; 10 )
// for loop creates a basic outline
 vCharacter := NTOC(96) + x
Tab()
Type ( "(" + vCharacter + ")" )
Indent()
HardReturn()
EndFor
EndProcedure
```

Working with constant variables

Constant variables — also called "constants" — represent a value that cannot change during macro play. As such, constants must be initialized upon declaration, and their assigned value cannot change. Constants should be used sparingly, if at all.

Expressions are formed by using operators (see "Understanding operators" on page 25) to combine constants with other types of variables. For more information about constants, see "Understanding constants" on page 25.

Determining whether variables exist

You can use the `Exists` command to determine whether a variable exists — that is, whether it has been declared and initialized. The following sample code shows how to use the `Exists` command:

```
// Declare and initialize a variable's Name := "Fred"
// Use Exists to see if the variable still exists as a local variable
If ( Exists ( sName; Local! ) )
```

```
...
EndIf
...
Quit
```

The `Exists` command returns a value after checking the specified variable against the variable tables. The variable tables are checked in the following order: local, then global, then persistent.

If you specify a variable-table parameter for the `Exists` command, a value of `True` is returned if that variable is found in the specified variable table. If the variable is not found in the specified variable table, a value of `False` is returned. The following example illustrates this scenario:

```
If( Exists ( x ; Global!) = TRUE )
x := 147
Else
Global ( x := 147 )
EndIf
```

If you do not specify a variable-table parameter for the `Exists` command, one of the following values is returned:

- `NotFound!` or `0` — indicates that the variable does not exist in any variable table
- `Local!` or `1` — indicates that the variable exists in the local-variable table
- `Global!` or `2` — indicates that the variable exists in the global-variable table
- `Persistent!` or `3` — indicates that the variable exists in the persistent-variable table

The following example illustrates this scenario:

```
Persist( x := 3 )
If ( Exists( x ) = Exists.Persistent!)
MessageBox ( retVar; "Variable"; "This variable Exists in the _
Persist variable pool (" + Exists ( x ) + ")" )
EndIf
```

Discarding variables

You can use the `Discard` command to remove a variable from memory by deleting it from its associated variable table. The following sample code shows how to use the `Discard` command:

```
// Declare and initialize a variable
sName := "Fred"
...
// Free the memory used by vName
Discard ( sName )
// sName no longer exists and cannot be accessed
...
Quit
```

The `Discard` command searches the variable tables in the following order: local, then global, then persistent. If variables with the same name exist in different variable tables, you may need to use the `Discard` command multiple times, as in the following sample code:

```
While(Exists(VariableName))
Discard(VariableName)
EndWhile
```

Working with arrays

If you want, you can assign a collection of data to a single variable name by creating an "array." The *elements* in a PerfectScript array can be declared and initialized in the same ways as variables. Unlike other programming languages, however, PerfectScript lets you assign the elements in an array to different data types. PerfectScript arrays therefore provide a powerful way to control large amounts of data on one or more dimensions.

To use an array, you must first declare it and initialize its elements. When a macro is played, a run-time error is incurred for each array element that is not both declared and initialized.

The commands for declaring arrays are the same as for declaring variables: `Declare`, `Local`, `Global` and `Persist`. At declaration, an array requires the following items:

- an alphanumeric (case-insensitive) name that begins with a letter and is limited to 50 characters
- a subscript, marked in brackets (`[ ]`), that specifies how many array elements to create

The following commands declare a one-dimensional array that contains five elements:

- `Declare(aMyArray[5])` — declares a one-dimensional, five-element local array
- `Local(aMyArray[5])` — declares a one-dimensional, five-element local array

- `Global(aGlobalArray[5])` — declares a one-dimensional, five-element global array
- `Persist(aPersistArray[5])` — declares a one-dimensional, five-element persistent array

Every array contains a hidden element called `0`. This element stores the total number of elements in the array (not including itself), and an attempt to assign any other value to this element generates an error message. In the previous examples, the declared array actually includes six elements if you include element `0`.

The preceding examples declare an array but do not initialize its elements. Before you can use an array, you must individually initialize each array element. To initialize an array element, you must specify the array name; the subscript (or index) number of the array element, enclosed in brackets (`[ ]`); and the desired value for the array element. The following example illustrates how to initialize each element in an array after declaring the array.

```
// Declare a 5-element array
Declare ( aMyArray[5] )
// Initialize each of the five elements
aMyArray[1] := "One"
aMyArray[2] := "Two"
aMyArray[3] := "Three"
aMyArray[4] := "Four"
aMyArray[5] := "Five"
```

You can simplify the process of initializing array elements after declaring an array by using the following syntax:

```
Declare ( aMyArray[5]; nCount := 0 )
ForEach (x; {"One"; "Two"; "Three"; "Four"; "Five"})
nCount := nCount +1
aMyArray[ nCount ] := x
EndFor
```

If you want, you can initialize an array upon its declaration (in which case, the number of elements need not be specified). The following commands declare and initialize a one-dimensional array that contains five elements:

- `Declare(aMyArray[]:={"One";"Two";"Three";"Four";"Five"})` — declares and initiatlizes a one-dimensional, five-element local array
- `Local(aMyArray[]:={"One";"Two";"Three";"Four";"Five"})` — declares and initiatlizes a one-dimensional, five-element local array
- `aMyArray[]:={"One";"Two";"Three";"Four";"Five"}` — declares and initiatlizes a one-dimensional, five-element local array
- `Global(aGlobalArray[]:={"One";"Two";"Three";"Four";"Five"})` — declares and initiatlizes a one-dimensional, five-element global array
- `Persist(aPersistArray[]:={"One";"Two";"Three";"Four";"Five"})` — declares and initiatlizes a one-dimensional, five-element persistent array

PerfectScript arrays can have up to ten dimensions. A two-dimensional array is like a table with rows and columns: Each cell in the table is an individual element.

For declaring a multi-dimensional array, the syntax of the subscript operator (`[ ]`) is as follows: The number of dimensions is followed by a semicolon (`;`), which is followed by the number elements within each dimension. The following example shows how to declare a three-dimensional array in which each dimension has five elements:

```
Declare ( aMyArray[3;5] )
```

Each dimension can have up to 32,767 elements (depending on available memory), and each element can be individually accessed and initialized. For accessing and initializing an element in a multi-dimensional array, the syntax of the subscript operator (`[ ]`) is as follows: The dimension number of the element is followed by a semicolon (`;`), which is followed by the subscript (or index) number of the element within that dimension.

The following syntax specifies the first element in the first dimension:

```
aMyArray[1;1] := "1-1"
```

The following syntax specifies the third element in the second dimension:

```
aMyArray[2;3] := "2-3"
```

The following stynax specifies the fifth element in the third dimension:

```
aMyArray[3;5] := "3-5"
```

Multi-dimensional arrays, like one-dimensional arrays, can be initialized at their time of declaration. In this scenario, the number of elements in each dimension does not need to be explicitly stated because it is implied by the actual initialization of those elements.

In addition, the dimensions are separated by a semicolon (`;`). The following example illustrates this syntax:

```
aMyArray[] :=
{{"1-1"; "1-2"; "1-3"; "1-4"; "1-5"};  // First row
{"2-1"; "2-2"; "2-3"; "2-4"; "2-5"};  // Second row
{"3-1"; "3-2"; "3-3"; "3-4"; "3-5"} }  // Third row
```

PerfectScript provides a special form of initialization for multi-dimensional arrays. This form, called a *slice* (`...`), lets you initialize the elements on a single dimension by repeating the last-initialized element throughout that dimension. When initializing with a slice, you must fully initialize at least one row in each dimension to define the extent of the slice. The following example illustrates the syntax for using a slice:

```
// Declares a three-dimensional array (3x3x6) initializing all
elements with a slice
aMyArray[] :=
{ { {1;1;1;1;1;1};  // First dimension, first row
{1; ... };  // First dimension, second row, replicated with value 1
{1; ... } };  // First dimension, third row
{{2;2;2;2;2;2};  // Second dimension, first row
{2; ... };  // Second dimension, second row
{2; ... } };  // Second dimension, third row
{{3;3;3;3;3;3};  // Third dimension, first row
{3; 4, ... };  // Third dimension, second row, replicated with value 4
{3; 1; ... } } }  // Third dimension, third row, replicated with _
value 1
```

You can use the `Dimensions` command to return the following information about an array:

- total number of dimensions in the array
- total number of elements in the array
- number of elements in each dimension
- index range

In some cases, you must declare an array dynamically and therefore cannot be sure how many dimensions are contained in the array. The `Dimensions` command allows a macro to act dynamically by querying the size of an array:

```
aFiles[] := GetFileList()  // returns an array of random size
ForNext (x; 1; Dimensions( aFiles[]; 0 ) )
// Dimensions queries the array for the size
FileOpen ( aFiles[x] )
FooterA(Create!)
Type ( "McRae's Eat-a-Burger ")
SubStructureExit()
FileSave ( aFiles[x]; WordPerfect_60! )
Close()
EndFor
Function GetFileList()
...  // statement block that creates an array (sized) dynamically
Return ( ArrayOfFiles[] )
EndFunction
```

Please note that `MacroArgs[]` is a special PerfectScript array that contains values that are passed to the macro by the commands `Chain`, `Nest`, or `Run`. The following example illustrates the `MacroArgs[]` array.

```
// Macro: MAIN.WCM
// Include full path if macro not
// in default macros directory
Run("TSTMACRO"; {"x"; "y"; "z"})

// Macro: TSTMACRO.WCM
// Compile, then play MAIN.WCM
vElements = Dimensions (MacroArgs[]; 0)
If (vElements != 0)
ForNext (x; 1; vElements; 1)
MessageBox (z; "Element Values"; "MacroArgs[" + x + "] = " + _
MacroArgs[x])
EndFor
Else
Beep
```

```
MessageBox (z; "Error"; "No values passed"; IconExclamation!)
EndIf
// Result: x y z
```

Understanding constants

A constant — also called a constant variable — represents a value that cannot change during macro play.

By contrast, most variables represent a value that can change during macro play. For more information, see "Understanding variables" on page 10.

Constants must be initialized upon declaration, and their assigned value cannot change. You can declare and initiatlize a constant in the following way:

```
Constant ( WPCLASSNAME := "WordPerfect.8.32" )
```

A compile-time error occurs if a constant is misused in one of the following ways:

- if the constant is not initialized upon declaration — for example, if the declaration statement is missing an assignment operator (`:=`) or an assigned value (or both)
- if an attempt is made to assign a different value to a constant after it has been declared and initialized

You can set constants apart from other variables by giving them a name that appears entirely in capital letters. This naming convention is the generally accepted practice in C/C++ and other programming languages.

Understanding operators

Operators are used to combine variables (see "Understanding variables" on page 10) and constants (see "Understanding constants" on page 25) into expressions — and even to combine expressions into other expressions. In PerfectScript, operators can be either "unary" or "binary."

Unary operators are symbols or words that represent an operation on only one operand or expression. The following table lists the unary operators that are available in PerfectScript.

Operator	Action	Example and result
`+`	Multiplies an operand by +1	`+5` **Result:** `+5` (that is, `5 * +1`)
`-`	Multiplies an operand by -1	`-10` **Result:** `-10` (that is, `10 * -1`)
`NOT`	Inverts the result of relational and logical expressions	See "Understanding logical operators" on page 31.
`~`	Toggles a binary value (that is, converts `1` to `0`, and `0` to `1`)	See "Understanding bitwise operators" on page 35.

Binary operators are symbols or words that represent an operation on two operands or expressions. In the following example, the binary plus operator (`+`) adds the operands `3` and `4`, and the assignment operator (`:=`) assigns the result of the arithmetic expression `3 + 4` to variable `x`.

```
x := 3 + 4
```

All PerfectScript operators can be classified into the following functional categories:

- *assignment operators* — symbols that assign the value of a right-operand expression to a left-operand variable. For more information, see "Understanding assignment operators" on page 27.
- *arithmetic operators* — symbols or words that represent a mathematical operation on two operands. For more information, see "Understanding arithmetic operators" on page 27.
- *relational operators* — symbols that represent a relational operation on two operands, such that the operation result equals either true or false. For more information, see "Understanding relational operators" on page 28.
- *logical operators* — words that represent a logical relationship between conditions, or that invert a condition. For more information, see "Understanding logical operators" on page 31.
- *bitwise operators* — symbols that represent a bitwise operation on two integer operands. For more information, see "Understanding bitwise operators" on page 35.

When used together to form an expression, operators are evaulated by PerfectScript based on precedence level. For more information about operator precedence, see "Understanding operator precedence" on page 38.

For detailed information on each operator, please see the “Operators” topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Understanding assignment operators

Assignment operators are symbols that assign the value of a right-operand expression to a left-operand variable.

The following table lists the assignment operators that are available in PerfectScript.

Operator	Action	Example and result
`:=` or `=`	Assignment of a value to a variable	`x := "John Doe"` **Result:** The variable `x` equals the character string `John Doe`.

Understanding arithmetic operators

Arithmetic operators are symbols or words that represent a mathematical operation on two operands.

All arithmetic operators are binary operators.

The following table lists the arithmetic operators that are available in PerfectScript.

Operator	Action	Example and result
`*`	Multiplication	
`/`	Division	
`-`	Subtraction of numbers and reduction of strings	`vStr := "abcdefg" - "efg"` **Result:** vStr equals `"abcd"` (by reduction of strings)
`+`	Addition of numbers and concatenation of strings	`vStr := "abcd" + "efg"` **Result:** `vStr` equals `"abcdefg"` (by concatenation of strings)

Operator	Action	Example and result

`%`	Floating point modulus division — returns remainder of floating-point division	`x:=10.1 % 3`
		Result: `x` equals `1.1`
		`x := 9 % 3`
		Result: `x` equals `0`

`MOD`	Integer modulus division — returns remainder of integer division	`x := 10 MOD 3`
		Result: `x` equals `1`
		`x := 10.1 MOD 3`
		Result: error (because `MOD` cannot be used on real numbers)

`DIV`	Integer division — returns integer portion of integer division	`x := 10 DIV 3`
		Result: `x` equals `3`
		`x := 9 DIV 3`
		Result: `x` equals `3`
		`x := 9.1 DIV 3.5`
		Result: error (because `DIV` cannot be used on real numbers)

`**`	Exponentiation — raises a number to a power	`vResult = 2**3`
		Result: `vResult = 8` (that is, `2` to the power of `3`)
		`vResult = 4**2`
		Result: `vResult = 16` (that is, `4` to the power of `2`)

Understanding relational operators

Relational operators are symbols that represent a relational operation on two operands. The operation result equals `True` or `False`.

All relational operators are binary operators.

The following table lists the relational operators that are available in PerfectScript.

Operator	Action	Example and result

`>`	Greater than	`x := 10`
		`z := (x>5)`
		Result: `z` equals `True` (because `x` is greater than `5`)

`>=`	Greater than or equal to	`x := 10`
		`If (x>=10)`
		`Beep`
		`Else`
		`Quit`
		`EndIf`
		Result: The computer beeps because the result of expression `x >= 10` equals `True` (that is, `x` equals `10`). The `Else` statement is skipped, so the `Quit` command does not end the macro.

`<`	Less than	`x := 10`
		`z := (x<5)`
		Result: `z` equals `False` (becuase `x` is not less than `5`)

Operator	Action	Example and result

`<=`	Less than or equal to	`x := 20`
		`If (x<=10)`
		`Beep`
		`Else`
		`Quit`
		`EndIf`
		Result: The computer does not beep because the result of expression `x <= 10` equals `False` (that is, `x` is greater than `10`). The `Else` statement is played, so the `Quit` command ends the macro.

`=`	Equal to
	Note: Whereas `LIKE` is `True` regardless of whether the compared strings are identical in case, `=` is case-sensitive.
		`x := 10`
		`z := (x=5)`
		Result: `z` equals `False` (because `x` is not equal to `5`)
		`x := 10`
		`z := (x = 10)`
		Result: `z` equals `True` (becuase `x` is equal to `10`)
		`x := "Abc"`
		`z := (x = "Abc")`
		Result: `z` equals `True` (because `x` is equal to `"Abc"`)
		`x := "Abc"`
		`z := (x = "abc")`
		Result: `z` equals `False` (because `x` must be the same as `"Abc"`, including case)

`<>`	Not equal to	

Operator	Action	Example and result

`!=`	Not equal to	`x := 10`
		`z := (x!=5)`
		Result: `z` equals `True` (because `x` is not equal to `5`)
		`x := 12`
		`z := (x !=12)`
		Result: `z` equals `False` (because `x` is equal to `12`)

`IN`	Membership	`z := 3 IN {1; 2; 3}`
		Result: `z` equals `True`
		`z := 3 IN (1; 2; 4}`
		Result: `z` equals `False`
		`z := {1; 2; 3} IN {1; 2; 3}`
		Result: `z` equals `True`
		`z := {1; 2} IN {1; 2; 3}`
		Result: `z` equals `True`
		`z := {1; 2; 3} IN {1; 2; 4}`
		Result: `z` equals `False`

`LIKE`	Case-insensitive string equality **Note:** Whereas = is `True` only if the compared strings are identical in case, `LIKE` disregards case altogether.	`z := ("abc" LIKE "Abc")` **Result:** `z` equals `True`

Understanding logical operators

Logical operators are words that either invert one condition or represent a logical relationship between conditions. A condition is the result of a relational expression (see "Understanding relational expressions" on page 45).

Most logical operators are binary operators. The exception is `NOT`.

The following table lists the logical operators that are available in PerfectScript.

Operator	Action	Example and result

`NOT`	Inverts the result of a relational expression	*See also the detailed NOT examples that follow this table.*
		`x := 8`
		`If ((x<10) AND NOT (x=5))`
		`Beep`
		`EndIf`
		Result: The result of the logical expression equals `True` (because `x` is less than `10`, `AND NOT x` is equal to `5`, or `x` is not equal to `5`).

`AND`	Combines two relational expressions. Each expression must be `True` for the logical expression to be `True`.	`x := 1`
		`y := 2`
		`z := ((x = 1) AND (y = 2))`
		Result: `z` equals `True`. The logical `AND` expression is `True` because both the relational expressions `x = 1` and `y = 2` are `True`.

Operator	Action	Example and result

`XOR`	Combines two relational expressions. Only one expression can be `True` for the logical expression to be `True`. If both are true or both are `False`, the logical expression is `False`. The XOR function is also called "exclusive OR."	*See also the detailed XOR example that follows this table.*
		`x := 1`
		`y := 2`
		`z := ((x = 0) XOR (y = 2))`
		Result: `z` equals `True` (because only one relational expression is `True`)
		`z := ((x = 1) XOR (y = 2))`
		Result: `z` equals `False` (because both relational expressions are `True`)
		`z := ((x = 0) XOR (y = 1))`
		Result: `z` equals `False` (because both relational expressions are `False`)

`OR`	Combines two relational expressions. Only one expression needs to be `True` for the logical expression to be `True`. The OR function is also called "inclusive OR."	`x := 1`
		`y := 2`
		`z := ((x = 1) OR (y = 5))`
		Result: `z` equals `True`. The logical `OR` expression is `True` because the relational expression `x = 1` is `True`.

Here are some more detailed examples of `NOT`.

NOT Example	Result

`x := 5`
`y := (x<10)`
`z := NOT(x<10)`
	If `x` is less than `10`, `True` is assigned to `y` and `False` (that is, the inverted result of the expression `x<10`) to `z`.

NOT Example	Result

`x := 5`
`z := (x<10)`
`If (NOT(z))`
`Beep`
`Else`
`Quit`
`EndIf`
	The result of the expression `x<10` is assigned to variable `z`. The `Beep` command causes the computer to beep if the inverted value of `z` equals `True`. However, the inverted value of `z` equals `False`, so there is no beep; the `Else` statement is played, and the `Quit` command ends the macro.
	NOTE: For more information about `If` conditions, see "Understanding If conditions" on page 55.

`x := 5`
`If (NOT(x<10))`
`Beep`
`Else`
`Quit`
`EndIf`
	In this shorthand notation, the computer is instructed to beep if the inverted result of the expression `x<10` equals `True`. However, the result of `NOT(x<10)` equals `False`, so there is no beep; the `Else` statement is played, and the `Quit` command ends the macro.

`x := 5`
`If (x>10)`
`Beep`
`Else`
`Quit`
`EndIf`
	This example represents an alternative to using the `NOT` operator. If the result of the expression `x>10` equals `True`, the computer beeps. However, the result equals `False` (because the value of `x` is less than `10`), so the `Else` statement is played and the `Quit` command ends the macro.

Here is a more detailed example of `XOR`.

XOR Example	Result

`x := 1`
`y := 2`
`If ((x = 0) XOR (y = 2))`
`Beep`
`EndIf`
	In this shorthand notation, the result of the logical `XOR` expression equals `True` because the relational expression `x = 0` is `False` and the relational expression `y = 2` is `True`.

Understanding bitwise operators

Bitwise operators are symbols that represent a bitwise operation on two-integer operands.

Most bitwise operators are binary operators. The exception is bitwise NOT (~).

The following table lists the bitwise operators that are available in PerfectScript.

Operator	Action	Example and result
~	Toggles a binary value (that is, converts `1` to `0`, and `0` to `1`). Also called "bitwise unary NOT" — "unary" because it has a single complement.	`~-15` **Note:** The binary equivalent of `-15` is `1111111111110001`. **Result:** `14` (binary `0000000000001110`)

&	Results in `1` if both operand bits are `1`, `0` if one of the operand bits is not `1`, or `0` if both operands are `0`. Also called "bitwise AND."	`x:=1000&31`
		Note: The binary equivalent of `1000` is `1111101000` and of `31` is `0000011111`.
		Result: `x` equals `8` (binary `0000001000`)
		`x:=65535&535`
		Note: The binary equivalent of `65535` is `1111111111111111` and of `535` is `0000001000010111`.
		Result: `x` equals `535` (binary `0000001000010111`)

Operator	Action	Example and result

`|`	Results in `1` if either operand is `1`. Also called "bitwise inclusive OR."	`x:=1000|27`
		Note: The binary equivalent of `1000` is `1111101000` and of `27` is `0000011011`.
		Result: `x` equals `1019` (binary `1111111011`)
		`x:=65535|535`
		Note: The binary equivalent of `65535` is `1111111111111111` and of `535` is `0000001000010111`.
		Result: `x` equals `65535` (binary `1111111111111111`)

`^`	Results in `0` if operands match, `1` otherwise. Also called "bitwise exclusive OR (XOR)".	`x:=1000^40`
		Note: The binary equivalent of `1000` is `1111101000` and of `40` is `0000101000`.
		Result: `x` equals `960` (binary `1111000000`)
		`x:=65535^535`
		Note: The binary equivalent of `65535` is `1111111111111111` and of `535` is `0000001000010111`.
		Result: `x` equals `65000` (binary `1111110111101000`)

Operator	Action	Example and result

`<<`	Shifts bits left by the specified number of places. For example, specifying `1` place shifts all bits one place to the left (and inserts a `0` at the right end of the binary number, effectively multiplying the value by 2.	`x:=500<<1`
		Note: The binary equivalent of `500` is `0111110100`.
		Result: `x` equals `1000` (binary `1111101000`)
		`x:=65535<<1`
		Note: The binary equivalent of `65535` is `1111111111111111`.
		Result: `x` equals `131070` (binary `1111111111111110`)

`>>`	Shifts bits right by the specified number of places. For example, specifying `1` place shifts all bits one place to the right (and inserts a `0` at the left end of the binary number, effectively dividing the value by 2.	`x:=1000>>1`
		Note: The binary equivalent of `1000` is `1111101000`.
		Result: `x` equals `500` (binary `0111110100`)
		`x:=65535>>1`
		Note: The binary equivalent of `65535` is `111111111111111`.
		Result: `x` equals `32767` (binary `0111111111111111`)

`<<<`	Rotates bits left by the specified number of places. For example, specifying `1` rotates all bits one place to the left.	`x:=-2147450881<<<1` **Note:** The binary equivalent of `-2147450881` is `10000000000000000111111 111111111`. **Result:** `x` equals `65535` (binary `00000000000000001111111 111111111`)

Operator	Action	Example and result
`>>>`	Rotates bits right by the specified number of places. For example, specifying `1` rotates all bits one place to the right.	`x:=65535>>>1` **Note:** The binary equivalent of `65535` is `0000000000000000111111` `1111111111`. **Result:** `x` equals `-2147450881` (binary `1000000000000000111111` `1111111111`)

Understanding operator precedence

The following table explains operator precedence, which is used by PerfectScript to evaluate expressions.

Precedence level	Operators
1	• parentheses [( )] • unary minus (`-`) • unary plus (`+`) • bitwise not (`~`) • logical not (`NOT`)
2	• exponentiation (`**`)
3	• multiplication (`*`) • division (`/`) • modulus division (`%` or `MOD`) • integer division (`DIV`)
4	• addition (`+`) • subtraction (`-`)
5	• left shift (<) • right shift (>) • left rotation (<) • right rotation (>)

Precedence level	Operators
6	•equality (=) •inequality (!=) •“less than” state (<) •“less than” state or equality (<=) •“greater than” state (>) •“greater than” state or equality (>=) •membership (`IN`) •case-insensitive string equality (`LIKE`)
7	•bitwise and (&) •bitwise or (|) •bitwise XOR (^)
8	•logical and (`AND`)
9	•logical or (`OR`) •logical xor (`XOR`)
10	•assignment (`:=` or `=`)

The following rules apply to operator precedence:

- Operators with the same precedence are evaluated from left to right.
- Operators inside parentheses are evaluated before operators outside parentheses.
- Operators inside nested parentheses are evaluated from the innermost parentheses out.

Here are some examples that illustrate operator precedence.

Example	Result
`x := ((50 * 5 + 50) * 3 + 100)`	`x` equals `1000`: •`50 * 5 = 250` •`250 + 50 = 300` •`300 * 3 = 900` •`900 + 100 = 1000`
`x := ((50 * (5 + 50)) * 3 + 100)`	`x` equals `8350`: •`5 + 50 = 55` •`55 * 50 = 2750` •`2750 * 3 = 8250` •`8250 + 100 = 8350`

Example	Result
`x := ((50 * 5 + 50) * (3 + 100))`	`x` equals `30900`: •`50 * 5 = 250` •`250 + 50 = 300` •`3 + 100 = 103` •`300 * 103 = 30900`

Understanding expression types

By combining variables (see "Understanding variables" on page 10) and constants (see "Understanding constants" on page 25) with operators (see "Understanding operators" on page 25), you can form expressions for use in macro statements.

PerfectScript macros support the following expression types:

- *numeric expressions* — numeric variables or numeric constants, or a combination of the two as joined by a numeric operator. For more information, see "Understanding numeric expressions" on page 41.
- *measurement expressions* — variables or constants that contain a measurement value, or a combination of the two as joined by a numeric operator. For more information, see "Understanding measurement expressions" on page 41.
- *radix expressions* — values that combine a number with a character that identifies the "radix" for that number (that is, the base of its number system). For more information, see "Understanding radix expressions" on page 42.
- *character expressions* — variables or character constants (such as letters, digits, or keyboard symbols), or a combination of the two as joined by the plus operator (+), the minus operator (-), or a relational operator. For more information, see "Understanding character expressions" on page 43.
- *arithmetic expressions* — statements that represent arithmetic operations, or statements that contain two operands that are joined by an arithmetic operator. For more information, see "Understanding arithmetic expressions" on page 45.
- *relational expressions* — statements that represent a relational operation, or statements that contain two operands that are joined by a relational operator. For more information, see "Understanding relational expressions" on page 45.
- *logical expressions* — statements that represent logical operations, or statements that contain two relational expressions that are joined by a logical operator. For more information, see "Understanding logical expressions" on page 46.

- *bitwise expressions* — statements that represent bitwise operations, or statements that contain two operands that are joined by a bitwise operator. For more information, see “Understanding bitwise expressions” on page 46.

Command calls and function calls can be used in an expression if they return a value. For more information, see “Using calling statements in macros” on page 59.

Understanding numeric expressions

Numeric expressions are numeric variables or numeric constants — or a combination of the two as joined by a numeric operator.

Given that `x` equals `3`, the following examples are valid numeric expressions.

Example	Explanation
`x`	Variable that contains a numeric value
`5`	Numeric constant
`x * 5`	Expression that multiplies `x` by `5` **Note:** This expression is also an arithmetic expression.
`+5`	Unary plus constant
`-(x + 10)`	Unary minus expression, which negates the result of `x` plus `10`

Understanding measurement expressions

Measurement expressions are variables or constants that contain a measurement value — or a combination of the two as joined by a numeric operator. A measurement value is created by combining a number (which represents the desired number of units) with a character (which identifies the desired unit of measurement).

The available units of measurement, and their associated identifiers, are as follows.

Unit of measurement	Identifier
Inches	`"` or `i`
Centimeters	`c`
Millimeters	`m`

Unit of measurement	Identifier
Points (72 per inch)	`p`
WP units (1200 per inch)	`w`

You can add and subtract measurement expressions as you do numeric expressions (see "Understanding numeric expressions" on page 41). When an operation is performed on measurement expressions that have different units of measure, the right operand is converted to the type of the left-measurement operand.

Combining numeric expressions with measurement expressions can produce unexpected results.

You do not need to specify a unit of measure for command-measurement expressions that follow `DefaultUnits`.

If you do not specify a unit of measure for a measurement expression, and `DefaultUnits` has not been encountered, the default unit of measurement WP units (1200 per inch) is used.

Given that `z` equals `4i` (that is, 4 inches), the following examples are valid measurement expressions.

Example	Explanation
`5c`	Constant (5 centimeters)
`z`	Variable that contains a measurement value of 4 inches
`z * 10i`	Expression that multiplies `z` by `10i` (that is, 10 inches)
`-z`	Unary minus, which yields `-4i` (that is, negative 4 inches)

Understanding radix expressions

The *radix* is the base of a number system. Radix expressions contain a radix value, which is created by combining a number (which represents the number value) with a character (which identifies the radix).

A radix value must begin with a number. For this reason, you must place a zero before any hexadecimal numbers that begin with the letters `A` through `F`.

The available radix choices, and their associated identifiers, are as follows.

Radix	Identifier
16 (hexadecimal system)	`x` or `h`
8 (octal system)	`o`
2 (binary system)	`b`

The following examples are valid radix expressions.

Example	Result
`x := 1Ah`	`x` equals the hexidecimal value of 26
`x := 0Ah`	`x` equals the hexidecimal value of 10 **Note:** A numeric value of `Ah` is not valid because it does not begin with a number. Instead, you must use `0Ah`.
`x := 1111b`	`x` equals the binary value of 15
`x := 44o`	`x` equals the octal value of 36

Understanding character expressions

Character expressions are character variables or character constants (such as letters, digits, or keyboard symbols) — or a combination of the two as concatenated by the plus operator (+), separated by the minus operator (-), or compared by a relational operator (such as >).

A character constant that is enclosed in single quotation marks specifies an ASCII numeric value, as in the following examples.

Example	Result
`x := 'A'`	`x` equals `65`
`x := 'A' + 'B'`	`x` equals `131` (that is, `65 + 66`)

A character string must be enclosed in double quotation marks. If the string already contains double quotation marks, it must use a second set of double quotation marks, as in the following examples.

Example	**Result**
`x := "His name is "John"" Doe"`	`x` equals `His name is "John" Doe`
`x := """John Doe"""`	x equals `"John Doe"`

The following examples are valid character expressions.

Example	**Explanation**
`"John Doe"`	Character string
`z := "Joe " + "Doe"`	Expression that is assigned to variable `z` (such that `z` equals `Joe Doe`)
`x := z + ", Jr."`	Expression that is assigned to variable `x` (such that `x` equals `Joe Doe, Jr.`)

If you concatenate a character string and a number, the number is converted to a character string. If you concatenate a numeric character and a number, the numeric character is converted to a number and the two are added. For examples, see the table that follows.

Example	**Result**
`x := "A" + 1`	`x` equals `A1` (which is a character string)
`x := "1" + 1`	`x` equals `2` (which is a number)
`x := ("A" + (1 + 3))`	`x` equals `A4` (which is the result of a mathematical operation [1+3=4] being converted to a character string before being concatenated to A)
`x := ("A" + 1 + 3)`	`x` equals `A13` (which is a character string because the numbers converted and not added, due to operator precedence)
`x := (1 + 3 + "A")`	`x` equals `4A` (which is a character string because the numbers are added and then converted to a character string, due to operator precedence)

Understanding arithmetic expressions

Arithmetic expressions are statements that represent arithmetic operations, or statements that contain two operands that are joined by an arithmetic operator. The result of an arithmetic operation is a numeric value.

Example	Result
`x := 1 + 2`	`x` equals `3`
`x := 3 * 3`	`x` equals `9`
`x := "2" * 3 * 4`	`x` equals `24` (because `2` is converted to a number then multiplied)
`x := "A" * 2`	Error (becuase letters and numbers cannot be multiplied by each other)

Understanding relational expressions

Relational expressions are statements that represent a relational operation, or statements that contain two operands that are joined by a relational operator. The result of a relational operation is either `True` or `False`.

Given that `x` equals `5`, the following examples yield the described results.

Example	Result
`z := (x = 6)`	`z` equals `False` (because `5` is less than `6`)
`z := (x = 5)`	`z` equals `True` (because `5` equals `5`)
`z := ("Ab" > "Bb")`	`z` equals `False` (becuase `Ab` is less than, or comes before, `Bb`)
`z := ("Ab" != "Bb")`	`z` equals `True` (because `Ab` is not equal to `Bb`)

Given that `x` equals `"A"`, `y` equals `"B"`, and `z` equals `"a"`, the following expressions return `True` or `False` in variable `w`.

Example	Result
`w := (x < y)`	`w` equals `False` (becuase uppercase A is less than, or comes before, uppercase B)

Example	Result
`w := (x > z)`	`w` equals `True` (uppercase A is greater than, or comes after, lowercase a).

Understanding logical expressions

Logical expressions are statements that represent logical operations, or statements that contain two relational expressions that are joined by a logical operator. The result of a logical operation equals `True` or `False`.

Given that `x` equals `10`, `y` equals `5`, and `z` equals `20`, the following expressions return `True` or `False` in variable `w`.

Example	Result
`w := ((x <= y) AND (y <= z))`	`w` equals `True` (because both relational expressions are true)
`w := ((x = y) AND (y <= z))`	`w` equals `False` (because the first relational expression is false)
`w := NOT(y > z)`	`w` equals `True` (because `5` is not greater than `20`)

`w := ((x != 5) AND (y != 20) _`
`AND (z = 20))`
	`w` equals `True` (becuase all relational expressions are true)

`w := (((x = 5) AND (y = 20)) _`
`OR (z = 20))`
	`w` equals `True` (because the expression `z = 20` is true)

`w := (((x = 5) AND (y = 20)) _`
`OR _NOT (z = 20))`
	`w` equals `False` (because all relational expressions are false)

Understanding bitwise expressions

Bitwise expressions are statements that represent bitwise operations, or statements that contain two operands that are joined by a bitwise operator. The result of a bitwise operation is a numeric value.

Consider the following examples of bitwise expressions.

Bitwise operator	Example and result

Bitwise NOT (~)	`x := ~(-15)`
	Result: `x` equals `14` (complement of `1`)
	`x := ~(-15) + 1`
	Result: `x` equals `15` (complement of `2`)

Bitwise AND (&)	`x := 65535 & 535` **Result:** `x` equals `535`
Bitwise inclusive OR (|)	`x := 65535 | 535` **Result:** `x` equals `65535`
Bitwise XOR (^)	`x := 65535 ^ 535` **Result:** `x` equals `65000`
Bitwise shift left (<<)	`x := 65535 << 1` **Result:** `x` equals `131070`
Bitwise shift right (>>)	`x := 65535 >> 1` **Result:** `x` equals `32767`
Bitwise rotate left (<<<)	`x := -2147450881 <<< 1` **Result:** `x` equals `65535`
Bitwise rotate right (>>>)	`x := 65535 >>> 1` **Result:** `x` equals `-2147450881`

Using command statements in macros

A command statement consists of a macro command, which represents a single instruction (typically, an action) in a macro.

Understanding macro commands

PerfectScript provides access to two main types of macro commands: *product commands* and *programming commands*.

OLE object commands represent a third type of PerfectScript macro commands. Also called "a method," an OLE object command performs a task on an OLE object in a specific OLE Automation server. For more information, see "Understanding OLE Automation" on page 77.

Product commands perform functions that let you use WordPerfect Office features in your macros. Product commands can be specific to one WordPerfect Office application or common to all of them. Many product commands require you to specify parameters that determine settings for dialog boxes or other application features (such as the ruler).

Product commands that report information (that is, return a value) about the state of an application or feature are sometimes called *system variables*. In WordPerfect, system variables begin with a leading question mark (as in `?ColumnWidth`). In Presentations, system variables begin with a leading `Env` (as in `EnvPaths`).

Programming commands perform functions that let you direct the function of a macro by controlling how application features act and interact. For example, you can use programming commands to specify macro conditions (see "Using conditional statements in macros" on page 54), specify that part of a macro run several times (see "Using loop statements in macros" on page 57), invoke or jump to a specified subroutine (see "Using calling statements in macros" on page 59), and so on.

You can use a product command by itself to create a basic macro that performs a simple task within a WordPerfect Office application. For example, the following product command displays the fourth slide in the current slideshow in Presentations:

```
ShowSlide(Slide: 4)
```

However, you must use product commands and programming commands together if you want to create a more complex macro. For example, the following code uses the product commands `LineHeightDlg` and `LineSpacingDlg` with the programming commands `If`, `Else`, and `Endif` to determine which dialog box to display in WordPerfect. (The **Line Height** dialog box is displayed if `x` equals the value `"A"`, while the **Line Spacing** dialog box is displayed if `x` has any other value.)

```
If (x = "A")
LineHeightDlg
Else
LineSpacingDlg
Endif
```

Understanding macro-command components

All macro commands have a *name*, and most macro commands have one or more *parameters* (which are marked by separators). For a PerfectScript macro to work properly, its macro commands must be spelled correctly and must include all required parameters (and the necessary separators) in the correct order.

When you create a macro by recording it (see "Recording macros" on page 92), the correct syntax is automtically applied to all macro commands. However, when you create a macro by typing code (see "Writing and editing macros" on page 93), you must manually apply the correct syntax to all macro commands.

In addition, some macro commands can be used to return data from various sources. Such commands are said to have *return values*.

As previously mentioned, product commands that return a value about the state of an application or feature are sometimes called *system variables*.

For more information about the components of a macro command, see the following topics:

- Understanding command names
- Understanding parameters
- Understanding return values

Understanding command names

The name of a macro command (that is, the "command name") indicates which feature is activated by that command.

Sometimes, a name is all that is necessary to perform the complete action of a macro command. For example, `FileOpenDlg` is a complete macro command because the name itself contains enough information to complete the task of displaying the **Open File** dialog box in WordPerfect.

Understanding command-name syntax

Command names are not case-sensitive. Although many commands appear in mixed case, you can type them entirely in uppercase or lowercase if desired.

Most command names do not contain spaces. Exceptions include programming commands that call a subroutine, such as `Case Call` or `OnCancel Call`.

For information about calling statements, see "Using calling statements in macros" on page 59.

Understanding parameters

While a command name (see "Understanding command names" on page 49) specifies a feature, some tasks require more information than this feature name alone can provide. To capture the settings for a feature, some macro commands provide one or more parameters, which are passed to the macro compiler (or between statement blocks) to carry out the desired task. For example, the WordPerfect product command `Backup()` is associated with the Automatic Document Backup feature, which can be toggled by specifying a parameter, as in `Backup(State:On!)`.

The type of information that is required by a parameter is represented by a *data type*. Each parameter accepts a specific data type. The most common data type for programming commands is *Variable* (see "Understanding variables" on page 10), while the most common data types for a product command are *String* (which specifies sequence of characters), *Numeric* (which specifies a numeric value), and *Enumeration* (which specifies one fixed value from a list of possible values).

In the macro-command syntax, data types are displayed *in italicized text*.

Parameters of data type `Enumeration` provide a set list of enumerations from which to choose. These enumerations are identified by a trailing exclamation point (`!`). For example, the WordPerfect command `BoxCaptionRotation` provides the parameter `Rotation`, which provides the following enumerations: `Degrees90!`, `Degrees 180!`, `Degrees 270!`, and `None!`.

In the following example of a WordPerfect macro command, `Advance` is the command name. `Where` is a parameter of data type *Enumeration*, and it is assigned the enumeration `AdvanceDown!`. `Amount` is a parameter of type *Numeric*, and it is assigned a numeric value of `1.0"`. The resulting macro command instructs WordPerfect to advance the insertion point down by one inch.

```
Advance (Where: AdvanceDown!; Amount: 1.0")
```

Understanding parameter syntax

The parameters for a macro command must be enclosed in a set of parentheses [`( )`]. Inserting a space between the command name and the left parenthesis is optional. However, using both a left parenthesis and a right parenthesis is mandatory; omitting either parenthesis is a common error than can prevent a macro from compiling.

Some programming commands and system variables have no parameters. Their syntax is the command name alone. Examples include the PerfectScript command `Pause` and the WordPerfect command `?FeatureBar`.

Some product commands have no parameters. Their syntax is usually written with empty parentheses. An example is the WordPerfect command `PosScreenUp ()`.

Using parentheses is mandatory for user-defined functions and procedures. For more information about functions and procedures, see "Understanding subroutines" on page 59.

A parameter is separated from its value by a colon (`:`). Inserting a space between colon and value is optional.

Each parameter ends with a semicolon (`;`). When a macro command requires several parameters, they must be placed in the order shown (and separated by their trailing semicolons). Inserting a space after a semicolon is optional.

For macro commands that have a single parameter, using the trailing semicolon is optional.

If you omit an optional parameter, you must include its semicolon in the syntax to keep the parameters that follow in their correct positions. Consider the following WordPerfect command:

```
AbbreviationExpand (AbbreviationName:; Template: PersonalLibrary!)
```

This command can be shortened as follows:

```
AbbreviationExpand (; PersonalLibrary!)
```

If a macro command accepts repeating parameters, the series must be enclosed in a set of braces (`{ }`).

Let's consider an example of parameter syntax in action. The `MakeItFit` command for WordPerfect has two parameters: `TargetPage` and `Adjust`. These parameters must be enclosed in a set of parentheses and separated by a semicolon. `Adjust` is a repeating parameter, so its instances must be separated by a semicolon, and this series of `Adjust` parameters must be enclosed in a set of braces. Here is an example of the proper syntax for this macro command:

```
MakeItFit (TargetPage: 1; {Adjust: FitTopMargin!; _
Adjust: FitFontSize!;})
```

Some macro statements are too lengthy to fit into a single line of macro code. If your macro editor automatically inserts a hard return at the end of every line, you must insert an underscore character (_) at the end of each line that wraps. For information on specifying a macro editor, see "To specify settings for editing macros" on page 86.

One way to reduce the length of a macro command is to omit parameter names. For example, the WordPerfect command `InhibitInput (State: Off!)` works the same as `InhibitInput (Off!)`. Similarly, consider the WordPerfect command `GraphicsLineLength (Length:` *`Numeric`*`)`, which can be written as follows:

```
GraphicsLineLength (Length: 2I)
```

or

```
GraphicsLineLength (2I)
```

Understanding return values

Some macro commands let you retrieve data from various sources. For example, such commands can get the current date from the system, the current page number or document filename from an application, or a specific value from the Windows® registry. This information is usually returned as a return value. Many programming commands provide return values, as do some product commands.

WordPerfect returns this type of information primarily with system variables.

Handling return values

To handle a return value, you must assign it to a variable (see "Understanding variables" on page 10) or use it in an expression (see "Understanding expressions" on page 8). For example, the expression `vVariable := ?Name` assigns the return value of `?Name` (which represents the filename of the current WordPerfect document) to the variable `vVariable`.

The return value of a system variable is handled in the same manner as the return value of a macro command.

To ignore a return value, don't handle it. For instance, some macro commands both change the state of an option and return the previous state of that option. If you want to change the state of an option without returning its previous state, you can ignore the return value.

Evaluating to return values

Macro commands that return values (and system variables) are said to “evaluate to” their return value. For example, because `(2+2)` evaluates to `4`, you can use `(2+2)` in an expression rather that using `4`. Similarly, because the WordPerfect system variable `?Name` evaluates to the filename of the current document, you can use `?Name` in an expression rather than using the filename of the current document.

Consider a macro that opens a file, writes text to it, and then closes it. To close the file, you can use the PerfectScript command `CloseFile`; however, this command also returns `True` if the file closes successfully (and `False` otherwise). Because `CloseFile` evaluates to its return value, you can use the following syntax to both close the file (where `xxxx` is the ID number of the file) and check whether it closes successfully:

```
If (CloseFile (FileID: xxxx))
...(statements to execute if the file was successfully closed)...
Else
...(statements to execute if the file was not successfully closed)...
EndIf
```

Return values can be handled outside of the context of a command. However, for return values of data type *Enumeration*, the returned enumeration has no meaning unless it is associated with a command. For example, the enumeration `On!` has no meaning by itself, but when used in the context of a command parameter, it indicates that that parameter is turned on. For this reason, PerfectScript evaluates return values of type *Enumeration* to the name of the command, followed by a period, followed by the enumeration (that is, `command name.enumeration!`).

For example, the syntax for the programming command `Cancel` is as follows:

```
enumeration := Cancel (State: Enumeration)
```

The `Cancel` command determines how a macro responds to a `Cancel` condition. It also returns the previous `Cancel` state (`On!` or `Off!`). The following example sets the `Cancel` state to `On!`, stores the current state of the `Cancel` command in the variable `vVariable`, and types `Correct` in the current WordPerfect document:

```
Cancel (State: On!)
vVariable := Cancel ()
If (vVariable = Cancel.On!)
Type ("Correct")
EndIf
```

```
If (vVariable = "Cancel.Off!")
Type ("Not Correct")
EndIf
```

If the optional parameter is omitted, the `Cancel` state can be returned without changing it. In this scenario, the `Type` command is not executed because the expression in the second `If` statement assumes that the enumeration returned by the `Cancel` command is a string. (Although enumerations look like strings, they are not.)

Enumerations have numeric equivalents. In the preceding example, `vVariable` is also equal to `1`. If you were to follow the above example with the WordPerfect product command `Type(vVariable)`, the number `1` would be typed in the current document. The numeric equivalents of enumerations can change, so as previously mentioned, you must use the syntax `command name.enumeration!` to evaluate to return values.

Using assignment statements in macros

Assignment statements assign the value of an expression (see "Understanding expressions" on page 8) to a variable (see "Understanding variables" on page 10). The assignment operator (`:=` or `=`) assigns the value of a right-operand expression to a left-operand variable.

For more information about assignment operators, see "Understanding assignment operators" on page 27.

For example, the result of the following assignment statement is that `x` equals `John Doe`:

```
x := "John Doe"
```

The result of the following assignment statement is that `y` equals `5`:

```
y := 5
```

The result of the following assignment statement is that `z` equals the result of `3 + 4`:

```
z := 3 + 4
```

Using conditional statements in macros

Conditional statements execute a statement (or statement block) when a specified condition is met — that is, when an expression is true, or when a variable matches a constant.

You can use a conditional statement to present the user with a list of options.

Conditional statements include `Case`, `If`, and `Switch`. For more information about these conditions, see the following topics:

- Understanding Case conditions
- Understanding If conditions
- Understanding Switch conditions

Understanding Case conditions

A `Case` condition executes a `Label` statement when `Test` (that is, a user-defined variable) matches a constant value.

In the following example, `Label (Start)` is called if `Test` matches `1`. If `Test` matches `2`, then `Label (Next)` is called. If there is no match, then `Label (Other)` is called.

```
Case (Test; {1; Start; 2; Next}; Other)
...(other statements)...
Label (Start)
...statement block...
Label (Next)
...statement block...
Label (Other)
...statement block...
```

`Case Call` is a similar condition to `Case`. A `Case Call` statement expects a `Return` after a `Label` statement.

Understanding If conditions

An `If` condition uses an `If-Else-Endif` construction to execute a statement (or statement block) when an expression is true.

In the following example, the first statement block is executed if the expression `x = 5` is true (that is, if `x` equals `5`). If the expression `x = 5` is not true, the second statement block is executed. `Else` is optional.

```
If (x = 5)
...statement block...
Else
```

```
...statement block...
Endif
```

In the following example, the statement block is executed if `Expression` is true. If `Expression` is not true, the first statement after `Endif` is executed. (Note, then, that for this example to work, `Expression` must evaluate to either true or false.)

```
If (Expression)
...statement block...
Endif
```

Understanding Switch conditions

A `Switch` condition uses a `Switch-EndSwitch` construction to execute a statement (or statement block) when `<Test>` matches `<Selector>`.

In the following example, the statement block after `Caseof <Selector>` is executed if `<Test>` matches `<Selector>`.

```
Switch (<Test>)
Caseof <Selector>:
...statement block...
Caseof <Selector>:
...statement block...
Caseof <Selector>:
...statement block...
Default:
...statement block...
EndSwitch
```

The statement block for a `Switch` conditon can call a subroutine (see "Understanding subroutines" on page 59). If `Continue` follows a statement block, the next statement block is automatically executed.

By using a `Switch` condition, you can alter the sequential play of macro commands. For example, if the following pair of commands is used in a macro, the second command overrides the first (because the **Paint Brush** width is set to 25 pixels and subsequently changed to 75 pixels):

```
SetBrushWidth(BrushWidth: 25)
SetBrushWidth(BrushWidth: 75)
```

If you want the macro to choose between these `SetBrushWidth` commands, you can use a `Switch` condition. In the following example, a **Paint Brush** width of `25` pixels is set if variable `Test` equals `1`. If `Test` equals `2`, a **Paint Brush** width of `75` pixels is set. Finally, if `Test` equals any value except `1` or `2`, a Paint Brush width of `50` pixels is set. (The value of `Test` can be determined by using a programming command such as `Menu` or `GetNumber`.)

```
Switch (Test)

Caseof 1: SetBrushWidth(BrushWidth: 25)

Caseof 2: SetBrushWidth(BrushWidth: 75)

Default: SetBrushWidth(BrushWidth: 50)

Endswitch
```

The following example contains two `CaseOf` statements. If variable `x` equals `1`, a subroutine named `Start` is called. If `x` equals `2`, a subroutine named `Stop` is called.

```
Switch (x)

CaseOf 1: CALL (Start)

CaseOf 2: CALL (Stop)

EndSwitch
```

Using loop statements in macros

Loop statements execute a statement (or statement block) a specified number of times until (or while) an expression is true. When the loop ends, the macro continues to the next statement.

You can indent lines to show levels of loop statements.

Loop statements include `For`, `Repeat`, and `While`. For more information about loop statements, see the following sections:

- Understanding For loops
- Understanding Repeat loops
- Understanding While loops

Understanding For loops

A `For` loop uses a `For-EndFor` construction to execute a statement (or statement block) a specified number of times.

In the following example, `<InitialValue>` initializes `<ControlVariable>`. `<TerminateExp>` tests the value of `<ControlVariable>`. `<IncrementExp>` increases the value of `<ControlVariable>` until `<TerminateExp>` is false and the loop ends. (If `<TerminateExp>` is initially false, then the statements do not execute because the test is checked at the start of the loop.)

```
For (<ControlVariable>; <InitialValue>; <TerminateExp>;
<IncrementExp>)

...statement block....

EndFor
```

In the following example, `x` is initialized to `1`. The statement block executes while `x` is less than `5`, and `x` is incremented by `1` at the end of each loop.

```
For(x; 1; x < 5; x + 1)

...statement block...

EndFor
```

Similar loop statements to `For` are `ForEach` and `ForNext`.

Understanding Repeat loops

A `Repeat` loop uses a `Repeat-Until` construction to execute a statement (or statement block) until an expression is true. All `Repeat` statements execute at least once because the expression is checked at the end of the loop.

In the following example, the statement block is executed until the expression `x = 10` is true (that is, until `x` is greater than or equal to `10`).

```
Repeat

...statement block...

Until (x >= 10)
```

However, the loop in the previous example does not end until the value of `x` changes to make the expression true. In the following example, the expression `x := x + 1` is used to increment `x` by `1` at the end of each loop so that the loop ends when `x` is greater than or equal to `10`.

```
Repeat

...statement block...

x := x + 1

Until (x >= 10)
```

Understanding While loops

A `While` loop uses a `While-EndWhile` construction to execute a statement (or statement block) while an expression is true. A `While` statement cannot execute unless the expression is true because the expression is checked at the start of the loop.

In the following example, the statement block is executed while the expression `x<= 10` is true (that is, while `x` is less than or equal to `10`). If `x` is greater than `10`, the loop does not execute.

```
While (x <= 10)
...statement block...
EndWhile
```

However, the loop in the previous example does not end until the value of `x` changes to make the expression true. In the following example, the expression `x := x + 1` is used to increment `x` by `1` at the end of each loop so that the loop executes while `x` is less than or equal to `10`.

```
While (x<=10)
 ...statement block...
x := x + 1
EndWhile
```

Using calling statements in macros

Calling statements involve a *subroutine*, which is one or more statements that are grouped as one item.

The larger a macro becomes, the more likely the need to create subroutines. Creating subroutines makes it easier to reuse code, and makes the macro easier to read and understand.

Understanding subroutines

A subroutine consists of a statement or a statement block that is played when called by a macro. Subroutines are useful because their statements are accessible to any part of a macro and can be called any number of times during play.

Consider the following example:

```
Call (SubExample)
...(other statements)...
```

```
Label (SubExample)
...statement block...
Return
```

In the preceding example, the calling statement `Call (SubExample)` calls (that is, directs macro play to) the subroutine `Label (SubExample)`. The `Return` command directs macro play to the statement that follows `Call (SubExample)`.

PerfectScript macros support the following types of subroutines:

- *labels* — act as a place holder, or marker, in a macro. For more information, see "Understanding labels" on page 60.
- *functions* and *procedures* — contain one or more statements that execute when called. Functions can be used to return a value, but procedures cannot. For more information, see "Understanding functions and procedures" on page 61.
- *callbacks* — enable a macro to respond immediately, and in specific ways, to events. For more information, see "Understanding callbacks" on page 74.

You can use subroutines to create calling statements. For more information, see "Creating calling statements from subroutines" on page 75.

Understanding labels

A label is a subroutine that acts as a place holder, or marker, in a macro. A macro can call the label when a certain function needs to be performed. After that function is performed, the `Return` command redirects execution to the command that immediately follows the call to the label.

Labels in the main body of a macro can execute without being called.

Labels cannot hide macro code or macro variables.

In general, macro labels are not used in structured programming unless they are needed within a function or procedure.

Creating labels

A label is created by using the `Label` command, which has one parameter: the name of the subroutine. The `Label` command takes no optional parameters.

Label names have the following conventions:

- They must begin with a character.
- They must consist of one or more letters or numbers.

- They are limited to 30 characters. (If a label name is longer than 30 characters, only the first 30 characters are recognized.)
- They have an optional trailing `@` sign.

Labels generally include one or more statements and are followed by the commands `Return` or `Quit`.

Calling labels

Label statements execute in the same way as other macro statements and do not need to be called. However, if desired, you can call a label by using any of the following PerfectScript commands:

- `Call`
- `Go`
- `Case`
- `Case Call`
- `OnCancel`
- `OnCancel Call`
- `OnError`
- `OnError Call`
- `OnNotFound`
- `OnNotFound Call`
- `OnDdeAdvise Call`
- `DdeExecuteExt`

Structuring labels

The following is an example of a label:

```
Call( MyLabel@ )

Quit

Label( MyLabel@ )

MessageBox (nVar; ""; "The Label was called.")

Return
```

Understanding functions and procedures

Functions and procedures are subroutines that contain one or more statements that execute when called. Most functions and procedures have parameters that receive values from a calling statement. However, some functions and procedures have zero parameters, in which case, they perform like a `Label` statement (see “Understanding labels” on page 60) except that they cannot execute unless called by the macro.

The difference between functions and procedures is that functions can return a value whereas procedures cannot.

Creating functions and procedures

Functions and procedures can be placed anywhere in a macro, or in a *macro-library file* (see “Storing functions and procedures in macro libraries” on page 71). Functions begin with the word `FUNCTION` and end with `ENDFUNC`, while procedures begin with the word `PROCEDURE` and end with `ENDPROC`.

A function or procedure cannot be defined inside another subroutine.

When you create a function or a procedure, you must name it. Function names and procedure names have the following conventions:

- They must begin with a character.
- They must consist of one or more letters or numbers.
- They are limited to 30 characters. (If a function name is longer than 30 characters, only the first 30 characters are recognized.)
- They can (optionally) have a trailing `@` sign.

Functions accept any of the following:

- a `Return` statement that has no parameters (return `0`)
- a value contained in a variable that is the result of a function
- an enumerated type that asserts a `Cancel`, `Error`, or `Not Found` condition
- a value contained in a variable that is the result of a function operation

Procedures accept any of the following:

- a `Return` statement that has no parameters (which direct macro execution to the statement that follows the caller of a procedure)
- a `Return` statement that has one parameter (which is an enumerated type that asserts a `Cancel`, `Error`, or `Not Found` condition)

Functions and procedures can include `Label` statements that are not visible outside the function. However, a `Label` statement inside a function must not have the same name as a function or procedure.

Using variables in functions and procedures

Function variables and procedure variables are local (or “private”) to the function or procedure. A variable with the same name as a function variable or procedure variable can be used elsewhere in the macro without conflict.

Variables are discussed, in general, in the section "Understanding variables" on page 10. However, the following details apply to using variables in funtions and procedures:

- By default, variables are created as local variables. Local variables are not visible to subroutines unless declared as part of the subroutine. (In other words, local variables created outside of a subroutine cannot be used by that subroutine.) The reverse is also true: If a variable is declared inside a subroutine, that variable cannot be used or accessed outside of that subroutine. (The exception to this rule occurs when a local variable created inside a function is returned to the calling statement by using the `Return` command, or when a parameter is passed by reference.)
- Global variables can be accessed anywhere in the macro for the life of the macro. These variables can be accessed and modified within any function or procedure. They are also visible to and can be modified by a macro that is started by the macro that declared the global variable.
- Persistent variables, like global variables, can be accessed and modified at any time during macro execution. The major difference between persistent variables and global variables is that a persistent variable exists after the macro that declared it finishes execution. For example, if macro A declares a persistent variable named `nTestVar` and sets its initial value to `3`, this variable is not discarded when macro A completes execution; if you run macro B, and macro B attempts to use `nTestVar`, the value of `nTestVar` is still `3`.

To destroy a persistent variable, you must either use the `Discard` command or close the Macro Facility. Merge variables are persistent variables, so they can be used during macro execution and merge execution.

The scope of a variable refers to the portion of a macro in which a variable is accessible. According to scope rules, variables can be created with the same name if they do not hold the same scope. If the scope of same-named variables is the same, the contents of the original variable are modified by the next instance of that variable.

To understand how scope affects a macro, examine the following code:

```
nHardReturn := NTOC(0f90ah)
Global sVariable1 := "I’m the first"
sVariable2 := "I’m the second"
sVariable3 := ""
sVariable4 :="I’m the fourth"
sVariable5 := CreateVariable( sVariable2; &sVariable3; sVariable4 )
MessageBox ( nretVal; "Variable Values";
"sVariable1: " + sVariable1 + nHardReturn +
"sVariable2: " + sVariable2 + nHardReturn +
"sVariable3: " + sVariable3 + nHardReturn +
"sVariable4: " + sVariable4 + nHardReturn +
"sVariable5: " + sVariable5)
Quit
Function CreateVariable( sVariable2; &sVariable3; sVariable4 )
sVariable1 := "I'm Global, I changed"
sVariable2 := "I’m not going to change"
sVariable3 := "I finally got initialized"
sVariable4 := "I am not really the Fourth"
Return ( sVariable4 )
EndFunc
```

In the preceding example, the only modified values are `sVariable1` and `sVariable3`. `sVariable1` is global and `sVariable3` is passed to the function by address; they are the only values that the function can “see.” The other variables were not modified because they were not within the scope of the function.

In the preceding example, the variable `sVariable4` may be in question. When this variable is passed to the function, a copy of the variable is made inside the function. This variable contains the same content and the same name as the original `sVariable4`, but it can be seen only by the function. The content of this second variable is modified and returned, and assigned, to a new variable called `sVariable5`.

Now consider the following example:

```
Global ( X )
X = "My name is John Doe"
DoCount()
MessageBox ( retVal; "Variable"; X )
Quit
Procedure DoCount()
x = 1
ForNext ( y; 1; 5 )
```

```
x = x * 10
EndFor
EndProc
```

In the preceding example, variable `X` would be equal to `"My name is John Doe."` if not declared global. The variable `x` declared inside the `DoCount` procedure would have been local to that procedure and would not have modified the contents of the original variable. Such problems become very apparent in large macros that include many variables. (This example also illustrates the need to give variables names that are meaningful.)

Use global or persistent variables only when necessary. When variables are declared as global or persistent and are visible to all sections of a macro, they may be changed or altered in ways that lead to unexpected behavior in your macros.

Passing variables to functions and procedures

Sometimes, you must pass parameters to functions or procedures. There are two ways to pass parameters to subroutines: *passing variables by value* and *passing variables by reference*.

When a variable is passed by value as a parameter to a subroutine, a copy of the variable is made in a different location in memory under a different name.

When a variable is passed by reference as a parameter to a subroutine, the address of the variable is used to provide direct access to that variable. In this way, you can use the original variable inside the subroutine that you call. Using this method lets you reduce memory usage and use fewer global variables in your macro.

The `&` operator is used when passing a variable by address. This operator is called the "Address Operator," and it tells the function or procedure to create a reference for this variable. The Address Operator is required both in the procedure call and in the procedure parameter list.

Here is an example of passing the address of a variable to modify its value. Procedures, unlike functions, cannot return values; however, in this case, the original value of the variable is modified by passing the variable by address to the procedure.

```
nNumber := 10
ChangeNumber ( &nNumber )
MessageBox (retVal; "New Value"; "The variable nNumber has been _
modified. The new value is " + nNumber +".")
```

```
Quit
Procedure ChangeNumber( &nNum )
nNum := nNum + 45
EndProc
```

If the preceding macro code were compiled and run, the `MessageBox` would display a value of `55`.

A function can return only one value. Sometimes, however, you need a function to generate two values to be returned and used later in the macro. In the following example, two values must be returned or changed: `OriginalCount` and `vBMName`. You can return only one value by using the `Return` command, so the other value can be modified by passing the other variable by address.

```
nOriginalCount := 0
MessageBox ( retVal; "Original Value"; nOriginalCount )
ForEach ( sString; {"One"; "Two"; "Three"; "Four"; "Five"})
//The line below passes nOriginalCount by address
sBookMarkName := CreateBookMarkName( sString; &nOriginalCount )
Prompt( "BookMark Names"; "New bookmark name: " + sBookMarkName; _
NoButtons!)
Wait( 5 )
EndFor
EndPrompt
MessageBox ( retVal; "New Value"; nOriginalCount )
Quit
Function CreateBookMarkName( sInputString ; &nCount)
nCount := nCount + 1
sBMName := sInputString + "_" + nCount
Return ( sBMName )
EndFunc
```

By passing the variable `OriginalCount` by address to the `CreateBookMarkName` function, we can manipulate the original value of the variable without having to return the variable. When the function receives the variable, it does not make a copy but references the original variable declared at the beginning of the code.

Passing arrays to functions and procedures

Like normal variables, arrays can be passed to subroutines. (For more information, see "Working with arrays" on page 20.) If you are passing the entire array, you must assign a value to each array element; any undefined element will be identified as an error at run-time.

Arrays can be passed by address.

Consider the following example:

```
Declare aOldArray[10]
ForNext ( x; 1; 10 )
// initialize all elements
aOldArray[x] := x
EndFor
aNewArray[ ] := Test( aOldArray[ ] )
// aNewArray[] is assigned the returned value
Type( aNewArray[10] )
// aNewArray[10] equals 100
HardReturn()
Type( aOldArray[10] )
// aOldArray[10] equal to 10
Quit
Function Test( aTestArray[ ] )
ForNext( x; 1; 10 )
// multiply all elements by 10
aTestArray[x] := x * 10
EndFor
Return( aTestArray[ ] )
EndFunc
```

In the preceding example, if you precede the calling-statement parameter and the corresponding function parameter with an ampersand (&), `100` is returned in `aOldArray[10]`, and all of the elements in `aOldArray[]` are modified to contain the new values in `aNewArray[]`. The following code illustrates this change.

```
Declare aOldArray[10]
```

```
ForNext ( x; 1; 10 )
// initialize all elements
aOldArray[x] := x
EndFor
aNewArray[ ] = Test( &aOldArray[ ] )
// aNewArray[] is assigned the returned value
Type( aNewArray[10] )
// aNewArray[10] equals 100
HardReturn()
Type( aOldArray[10] )
// aOldArray[10] equal to 100
Quit
Function Test( &aTestArray[ ] )
ForNext( x; 1; 10 )
// multiply all elements by 10
aTestArray[x] := x * 10
EndFor
Return( aTestArray[ ] )
EndFunc
```

Calling functions and procedures

To create a calling statement from a function or procedure, you specify the name of the function or procedure and one or more parameters that contain values passed to the function or procedure. (If there are no parameters, empty parentheses must follow the function name or procedure name.)

The following rules apply to calling functions and procedures:

- Functions and procedures do not execute unless they are called. They can be called from within another subroutine, or they can be called recursively (that is, they can call themselves).
- The number of parameters in a calling statement must match the number of parameters in the function or procedure. When a function or procedure requires multiple parameters, use semicolons (`;`) to separate those parameters.

Structuring functions

The basic structure of a function is as follows:

```
Function MyFunction()
. . . statement block
EndFunction
```

The `Function` keyword is followed by the actual function name. Statements are added to the function, after which the function is ended by the `EndFunc` or `EndFunction` keyword. A function can return a value by means of the `Return` command, as follows:

```
nOriginalValue := 6
nNewValue := AddNumbers( nOriginalValue )
MessageBox( nretVal; "New Value"; "The old value was " +
nOriginalValue + ". The new value is " + nNewValue + ".";
IconInformation! )
Quit
Function AddNumbers( nInputValue )
nTempValue := nInputValue + 13
Return (nTempValue)
EndFunc
```

In the preceding example, `nOriginalValue` is initialized with a value of `6`. This value is passed as a parameter to the `AddNumbers` function. The `AddNumbers` function adds `13` to `nInputValue` and stores the result in `nTempValue`. `nTempValue` is returned to the calling statement as a parameter of the `Return` command. The return value is assigned to `nNewValue`. The values of the `nOriginalValue` and `nNewValue` variables are then displayed by using the `MessageBox` command.

In the preceding example, the value of `nOriginalValue` had to be passed as a parameter to the function. If the `AddNumbers` macro function had attempted to access `nOriginalValue` without passing the value to the function, an error would have occurred. This is because `nOriginalValue` was out of scope of the `AddNumbers` function.

Structuring procedures

The basic structure of a procedure is as follows:

```
Procedure MyProcedureName( )
. . . statement block
EndProc
```

The `Procedure` command begins the block of code, with the actual name of the procedure following. The end of the procedure is marked by the `EndProc` or

`EndProcedure` keyword. The contents of the subroutine are placed between the `Procedure` and `EndProcedure` commands. If a procedure requires parameters, the code is similar to the following:

```
Procedure MyProcedureName( value1; value2 )
. . . statement block
EndProc
```

To call the preceding procedure, you would use the following code:

```
MyProcedureName( 1; 2 )
```

An example of a procedure call without parameters would resemble the following:

```
Procedure CreateFooter()
FooterA (Create!)
FontSize(10p)
InsertFilenameWithPath ()
Tab()
DateText ()
FlushRight ()
PageNumberDisplay ()
SubstructureExit ()
EndProc
```

In the preceding example, the `CreateFooter` procedure creates a footer in a document. This footer contains the text for path and filename, the date, and a page number — all formatted by using the specified formatting codes. The procedure does not receive any parameters.

We could modify the `CreateFooter` procedure to accept a value for the font size:

```
CreateFooter (10)
// Calls the procedure CreateFooter with the FontSize of // 10
Quit
Procedure CreateFooter( nFontSize )
nFontSize := 16.6 * nFontSize
// This calculates the correct font size value in WP units
FooterA (Create!)
FontSize( nFontSize)
```

```
// Recalculated value used
InsertFilenameWithPath ()
Tab()
DateText ()
FlushRight ()
PageNumberDisplay ()
SubstructureExit ()
EndProc
```

In the preceding example, the call made by the `CreateFooter` command passes one value, `10`, to the procedure. This value is received into a procedure variable that is named `nFontSize`. This variable can be used and manipulated only inside the procedure. In the example, a calculation is made to determine the proper font height in WordPerfect units, and then the variable is used by the `FontSize` command. After this routine has ended, the variable is discarded by PerfectScript.

Storing functions and procedures in macro libraries

Macro libraries contain files that store functions or procedures (or both) that can be called from another macro and must be compiled.

The following example contains two functions. Function `Add` receives a value in variable `x`. `50` is added to `x` and returned to the caller of the function. Function `Subtract` receives a value in another variable named `x`. `25` is subtracted from `x` and returned to the caller of the function.

```
Function Add(x)
x := x + 50
Return (x)
EndFunc
Function Subtract(x)
x:= x - 25
Return (x)
EndFunc
```

The `Use` command lets you use functions and procedures that are stored in another macro. This command usually precedes calling statements to a macro library.

If the preceding example were saved and compiled as `LIBRARY.WCM`, its `Add` and `Subtract` functions could be called as in the following example. After function `Add` is called, `100` is returned in variable `z`. After function `Subtract` is called, `75` is returned in variable `z`. The computer then beeps because the expression `z = 75` is true.

```
Application (WP; "WordPerfect"; Default; "EN")
Use ("C:\...\LIBRARY.WCM")
z := Add(50)
z := Subtract(z)
If (z = 75)
Beep // computer beeps because z equals 75
EndIf
```

The preceding example can be written in shortand notation, as follows:

```
Application (WP; "WordPerfect"; Default; "EN")
Use ("C:\...\LIBRARY.WCM")
z := Add(Subtract(50))
If (z = 75)
Beep // computer beeps because z equals 75
EndIf
```

`Use` is a non-executing statement that can occur anywhere in a macro. A macro that makes a call to a function or procedure in another macro file must include a `Use` statement that identifies the file.

A macro library that includes only function statements or procedure statements (or both) must be compiled like any macro file. PerfectScript automatically compiles uncompiled libraries. Macro execution stops if the macro library file will not compile.

Many programmers create library files that contain just functions and procedures that may be used with other macros. These functions and procedures are generic enough to be applicable to many different macros.

Here is an example of a macro that uses a function from another macro:

```
Macro1
Use( "Library.wcm" )
vDefDir := GetMyDefaultDirectory()
MessageBox ( retVal; "Docs Directory";
vDefDir" )
Quit
```

```
Library.wcm (macro library that contains the following code)
Function GetMyDefaultDirectory()
Return ( ?PathDocument )
EndFunc
```

If one macro "uses" another macro, the second macro becomes a dependent of the main macro. If you want to deploy the main macro throughout your organization or send it to a customer, You must include the second macro.

Macro files included by using multiple `Use` commands are searched from beginning to the end of the macro. Thus, a parent macro always calls the first occurrence of a function or procedure with the same name in different `Use` files. Consider the following example:

```
Macro1
Use( "Library1.wcm" )
Use( "Library2.wcm" )
vDefDir := GetMyDefaultDirectory()
MessageBox ( retVal; "Docs Directory";
vDefDir" )
Quit
Library1.wcm (contains)
Function GetMyDefaultDirectory()
Return ( ?PathDocument )
EndFunc
Library2.wcm (contains)
Function GetMyDefaultDirectory()
Return ( ?PathCurrent )
EndFunc
```

In the preceding example, the library function called by the macro is `Library1.wcm` because it is the first macro library included by `Macro1`.

When a macro uses a subroutine in a macro library, playing that macro incurs an error if the syntax of the call to the subroutine is incorrect. Using a function prototype or procedure prototype forces the compiler to check the parameter count of a function or procedure in a macro library; if the syntax is incorrect, a compiler error occurs.

The prototype directs the compiler to validate the syntax of a function or procedure. Creating a prototype helps you keep track of the parameters of a function or procedure. Place prototypes at the beginning of your macro, as in the following example:

```
Function Prototype Check(nBeep; HdReturn)
HdReturn := NTOC(0F90Ah)
x := 4
nBeep := 1
While(x = 4)
BEEP
x := Check(nBeep; HdReturn)
nBeep := nBeep + 1
EndWhile
MessageBox( retVal; "RETURN"; "The value of variable vStatus (" + x _
+ ") is returned to variable x, which ends the loop."; _
IconExclamation!)
Function Check(nBeep; HdReturn)
MESSAGEBOX(vStatus; "FUNCTION EXAMPLE"; "Beeps: " + nBeep + _
HdReturn + HdReturn + "Choose Retry to beep again." + HdReturn; _
IconInformation! | RetryCancel!)
RETURN(vStatus)
EndFunc
```

Understanding callbacks

Callbacks are special functions that enable a macro to respond immediately, and in specific ways, to events. When a macro executes a callback routine, the macro system automatically creates variables that are accessible to that callback. The callback can access the parameter array for information about the callback event, and the callback can place its return value (if any) in the return variable.

The parameters for a callback are placed into a global array variable that has the same name as the callback label. Any return value for the callback routine is placed in a (non-array) global variable that has the same name as the callback label. For example, parameters would be passed to callback routine

`'MsgHandler'` in an array called `'MsgHandler[ ]'`, and any return value would be placed into a variable called `'MsgHandler'`.

PerfectScript currently supports three types of callbacks, all of which are discussed in greater detail in the “Support for callback entries” topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**):

- product-command callbacks
- dialog-box callbacks
- message-box callbacks

Perhaps the most useful of these callback types is the dialog-box callback, which lets the macro gather information from an active dialog box (rather than waiting until the dialog box is closed to gather that information). For more information on dialog-box callbacks, see “Setting up callbacks for dialog boxes” on page 124.

Creating calling statements from subroutines

You can create calling statements by using PerfectScript commands, such as the following:

- `Call` — calls the specified subroutine. For more information, see “Using the Call command in calling statements” on page 75.
- `Go` — jumps to the specified subroutine. For more information, see “Using the Go command in calling statements” on page 76.
- `Case` or `Case Call` — creates a conditional statement that tests for matching expressions, and calls a label if a match is found. For more information, see “Understanding Case conditions” on page 55.

For more information about these (and other) PerfectScript commands, please see the PerfectScript Command Reference in the PerfectScript Help file (**psh.chm**).

Using the Call command in calling statements

The `Call` command has one parameter, which is the name of a subroutine to call. The `Return` command directs macro execution to the statement that follows `Call`.

In the following example, `Call(ExSub)` directs macro execution to `Label(ExSub)`, where the statement block is executed. `Return` directs macro execution to the first statement after `Call(ExSub)`.

```
Call (ExSub)

...other statements...
```

```
Label (ExSub)
...statement block...
Return
```

Using a subroutine name to form a calling statement performs the same action as creating a `Call` statement that specifies that subroutine name as a parameter. For example, a function or procedure that is named `InitializeVariables` can be called as follows:

```
Call InitializeVariables (<Parameter>; <Parameter>)
```

or

```
InitializeVariables (<Parameter>;<Parameter>)
```

If the second example calls a function, you can assign a return value to a variable with a statement such as the following:

```
x := InitializeVariables (<Parameter>;<Parameter>)
```

For information about return values, see “Understanding return values” on page 52.

For information about variables, see “Understanding variables” on page 10.

Using the Go command in calling statements

The `Go` command has one parameter, which is the name of a subroutine to which to jump. Macro execution continues from the point of the subroutine and does not return (so statements between `Go` and the subroutine do not execute). `Return` ends a macro or directs macro execution to the statement that follows a `Run` command.

In the following example, `Go(ExSub)` directs macro execution to `Label(ExSub)`, where the statement block is executed. `Return` ends the macro.

```
Go (ExSub)
...other statements...
Label (ExSub)
...statement block...
Return
```

Using comment statements in macros

Comment statements contain notes and other information that do not affect macro play. You can use comment statements to explain the purpose of your macro, describe its components, or to prevent a statement from playing.

A comment statement can consist of a single line of text or instead span several lines of text. However, the syntax for a single-line comment statement is different from that of a multi-line comment statement:

- single-line comment statement — begins with `//` and ends with a hard return
- multi-line comment statement — begins with `/*` and ends with `*/`

Accessing external applications in macros

PerfectScript provides the following advanced features, which let macros access applications outside of WordPerfect Office:

- *OLE Automation* — lets PerfectScript control applications that support OLE. For more information, see "Understanding OLE Automation" on page 77.
- *Dynamic Data Exchange (DDE)* — lets PerfectScript control applications that support DDE. For more information, see "Understanding Dynamic Data Exchange (DDE)" on page 80.

Understanding OLE Automation

PerfectScript can send commands that control WordPerfect, Quattro Pro, and Presentations. However, through a Windows-standard interface known as *OLE Automation*, PerfectScript can send commands that control other OLE-enabled applications, which are called OLE Automation servers. For this reason, PerfectScript is called an OLE Automation controller.

OLE Automation servers define OLE Automation objects, which have names that are registered with Windows. (For information about the OLE Automation objects that are defined for an application, refer to the manufacturer's documentation for that application.)

OLE Automation objects can have *methods* and *properties*.

A method is a command or function that performs an action on an object. Many methods, like product commands, have parameters and return values. Here is a sample method for the OLE Automation object **Excel.Application**:

```
Worksheets ().Activate
```

A property is an object value that can be retrieved and set. Many properties have parameters and return values. Unlike WordPerfect system variables, many properties can be set by placing the property name on the left side of an assignment statement (see “Using assignment statements in macros” on page 54). Properties can take parameters when being retrieved (similarly to a method call) or when being set. Here is a sample property for the OLE Automation object **Excel.Application**:

```
ActiveSheet.Name
```

IMPORTANT: You must set optional parameters when using OLE Automation.

Working with OLE Automation objects

Before an OLE Automation object can be used in a PerfectScript macro, the `Object` statement must be used. The `Object` statement is similar to the `Application` statement for products: It defines an object-prefix variable that is used to call methods and to retrieve and set object properties. The name of the object is also specified, along with information about whether this prefix is to be used as the default object for non-prefixed methods and properties.

The object-prefix variable that is specified in the `Object` statement identifies a variable that contains an instance handle to the object at run-time. As a variable, this prefix can be used in many places where most, but not all, other macro variables can be used. For example, the macro language operators `+` and `-` cannot be used with object variables, and automatic type conversions are not defined for object variables. Object variables can be assigned to other object variables of the same type, and they can be passed as parameters to user-defined macro routines.

As with other macro variables, object variables exist at run-time in a specific macro variable pool. You can specify this pool by using `Declare`, `Local`, `Global` and `Persist` statements (see “Understanding variables” on page 10). If the object variable is not specified, it exists in the local variable pool (unless `PersistAll` is in effect, in which case the object variable exists in the persistent variable pool).

Before making a call to the methods for an object, and before retrieving or setting the properties for an object, an instance handle to a specific object must be obtained through the `CreateObject` statement or the `GetObject` statement. These statements return an instance handle to a specific instance of an object. Because many OLE Automation servers support multiple instances of the objects they define, an instance handle to a specific object must be obtained to distinguish instances of the same object. Multiple object variables can be created, and multiple instances can be obtained if the OLE Automation server supports multiple instances.

After getting an instance handle, an object variable is said to be connected to the OLE Automation server. Like other variables, the `Exists` statement can be used on object variables to determine whether an object variable exists and, if so, which variable pool it exists in. Even if an object variable exists, it may not be currently connected to the OLE Automation server, but this connection information can be obtained from the `ObjectInfo` command.

When an instance of an OLE Automation object is no longer needed, the connection can be terminated by using the `Discard` statement. Variables for OLE Automation objects, like other variables, are automatically discarded when the macro terminates or when the user-defined routine in which the variable is defined ends. This discarding automatically disconnects the object from an OLE Automation server (if connected to one).

After an object is connected to its OLE Automation server, the methods and properties of that object can be accessed by prefixing the method name or property name with the variable name for the OLE Automation object followed by a period (`.`). If the OLE object variable is the default object variable (specified by `Default!` in the `Object` statement, or specified in a `With` statement), the object-prefix variable can be replaced by two leading periods (`..`). Leading periods are necessary to inform the macro compiler that the method name being called is not the name of a user-defined macro routine but the method name (or property name) of the current default OLE Automation object.

A macro can use more than one application product and OLE Automation object. Commands to the non-default application or OLE Automation object require a prefix, which is specified in a PerfectScript `Application` or `Object` statement. In the example `A1.AboutDlg ()`, the prefix `A1.` tells the compiler to use the application or object that is assigned `A1` in a PerfectScript `Application` or `Object` statement.

To establish a new default object for a localized block of code, you can use a **With-EndWith** compound statement. The object-prefix variable is specified in the `With` statement; all statements to access methods, and properties preceded by two periods (..) until the `EndWith` statement, are assumed to be references to that object.

The `NewDefault` statement can be used to establish a new default object-variable prefix for the remainder of a macro (or until the macro encounters another `NewDefault` statement).

For more information about PerfectScript support for OLE Automation, please see the "Support for OLE Automation" topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Understanding Dynamic Data Exchange (DDE)

Dynamic Data Exchange (DDE) Execute is a Windows feature that enhances product integration by allowing applications to instruct each other to perform specific tasks (that is, to execute commands). For example, you can use the DDE Execute feature to create macros in WordPerfect that send commands to control other Windows applications that accept DDE Execute strings.

WordPerfect, Quattro Pro, and Presentations are DDE servers, and they provide support to DDE clients through the PerfectScript language.

An instance of DDE-based communication between two applications is called a *DDE conversation*. WordPerfect, Quattro Pro, and Presentations each handle DDE conversations in their own way.

For more detailed information about using DDE, see the WordPerfect Office Software Development Kit (SDK), which is included in the Professional Edition of WordPerfect Office. For more information, see "Using the WordPerfect Office Software Development Kit (SDK)" on page 81.

Understanding DDE conversations for WordPerfect

WordPerfect can act as a server for `DDEExecute` commands.

To access the PerfectScript-based product commands for WordPerfect, the DDE client must initiate a DDE conversation by using `WPWin15_Macros` as the service name and `Commands` as the topic name.

Understanding DDE conversations for Quattro Pro

Quattro Pro can act as a server for `DDERequest` commands.

To access request topics, application status, or properties from Quattro Pro, the DDE client must initiate a DDE conversation by using `QPW` as the service name and `System` as the topic name.

To read and write to spreadsheet cells, or to execute PerfectScript-based product commands for Quattro Pro, the DDE client must initiate a DDE conversation by using `QPW` as the service name and the path and file of an open notebook as the topic name.

Understanding DDE conversations for Presentations

Any eligible Windows application can use the DDE Execute feature to control Presentations. To begin, the DDE client must initiate a DDE conversation by using `Presentations` as the service name and `Command` as the topic name. Before terminating the conversation, the client can send DDE Execute strings that include PerfectScript-based product commands for Presentations (provided that those commands do not contain variables or expressions).

Eligible Windows applications can also send commands to Presentations as a DDE Request item. In this scenario, Presentations returns an ANSI® text string that represents the return value of the command.

The DDE client must send DDE Execute strings and DDE Request items in ANSI text format. If Presentations returns an error, the DDE client can determine what went wrong by sending a DDE Request item for `LastCmdError`; Presentations then returns an ANSI text string that contains a three-digit error code and a description of the error.

Learning more about macros

If you want to learn more about macros, you can consult the WordPerfect Office Software Development Kit (SDK) or the Corel Web site.

This section contains the following topics:

- Using the WordPerfect Office Software Development Kit (SDK)
- Using the Corel Web site

Using the WordPerfect Office Software Development Kit (SDK)

The WordPerfect Office Software Development Kit (SDK) is a set of tools that lets you customize WordPerfect Office applications for commercial or business use. The WordPerfect Office SDK includes documentation, samples, and various tools and utilities.

The WordPerfect Office SDK is included with certain editions of WordPerfect Office.

Using the Corel Web site

The Corel Web site provides additional information about WordPerfect Office and about the products and services that are offered by Corel Corporation. Visit the Corel Web site at www.corel.com, or access it directly from WordPerfect, Quattro Pro, or Presentations by clicking **Help ▸ Corel on the Web** and then choosing a destination.

Getting started with macros

Now that you understand the basics about PerfectScript, you are ready to get started with macros by learning how to use the PerfectScript utility.

This section contains the following topics:

- Using the PerfectScript utility
- Specifying PerfectScript settings

Using the PerfectScript utility

To get started with PerfectScript macros for WordPerfect Office, you can use the PerfectScript utility.

WordPerfect, Quattro Pro, and Presentations provide a **Tools ▸ Macro** menu, which lets you work with macros from directly within the application. (WordPerfect also provides a **Tools ▸ Template macro** menu, which lets you work with template macros and QuickMacros from directly within WordPerfect.) For information about working with macros from directly within WordPerfect, Quattro Pro, or Presentations, please see the Help file for the application.

The PerfectScript utility provides the following tools for creating PerfectScript macros:

- **Command Browser** — displays a list of all available programming commands for PerfectScript, as well as all available product commands for WordPerfect, Quattro Pro, and Presentations.
- **Dialog Editor** — lets you create dialog boxes for your macros.

The PerfectScript utility provides context-sensitive Help for many of its controls, as well as Help for all of the macro commands in its Command Browser.

You can quit the PerfectScript utility when you have finished using it.

This section contains the following procedures:

- To start the PerfectScript utility
- To display the Command Browser
- To display the Dialog Editor

- To access context-sensitive Help for the PerfectScript utility
- To access Help for a macro command
- To quit the PerfectScript utility

To start the PerfectScript utility

- Click **Start** ▸ **All programs** ▸ **WordPerfect Office nn** ▸ **Utilities** ▸ **PerfectScript**, where **nn** is the version number of the software.

To display the Command Browser

- In the PerfectScript utility, click **Help** ▸ **Macro Command Browser**.

For information about using the Command Browser to create macros, see "Writing and editing macros" on page 93.

To display the Dialog Editor

- In the PerfectScript utility, click **Tools** ▸ **Dialog Editor**.

The Dialog Editor works only with macros that are in WordPerfect format. For this reason, you can open the Dialog Editor from directly within WordPerfect by clicking **Dialog Editor** on the **Macro** toolbar when editing a macro.

For information about using the Dialog Editor, see "Creating dialog boxes for macros" on page 101.

To access context-sensitive Help for the PerfectScript utility

- Click (or press **Shift + F1**), and then click the desired control.

To access Help for a macro command

- In the Command Browser, right-click the desired command.

To quit the PerfectScript utility

- Click **File** ▸ **Exit**.

Specifying PerfectScript settings

From within the PerfectScript utility, you can specify various PerfectScript settings.

This section contains the following procedures:

- To specify general macro settings
- To specify settings for compiling macros
- To specify settings for debugging macros
- To specify settings for editing macros
- To specify settings for playing macros
- To specify settings for recording macros
- To specify settings for the PerfectScript toolbar

To specify general macro settings

1 Click **Tools** ▸ **Settings**, and then click the **General** tab.

2 Do any of the following:
   - Specify a default macro folder.
   - Enable the **Use enhanced file dialogs** check box if you want to view detailed dialog-box information.
   - Enable the **Display icons in system tray** check box if you want to display macro icons in the Windows system tray.
   - Enable the **Check file associations on startup** check box if you want to check file associations at startup.

Click **Reset all to defaults** to return all settings to their original state.

To specify settings for compiling macros

1 Click **Tools** ▸ **Settings**, and then click the **Compile** tab.

2 Enable any of the following check boxes:
   - **Show progress**
   - **Include debug information**
   - **Warn when using unsupported features**
   - **Generate listing file**

To specify settings for debugging macros

1 Click **Tools** ▸ **Settings**, and then click the **Debug** tab.

2 Do any of the following:
- Enable the **Invoke Debugger on macro start** check box, if you want.
- Enable the **Invoke Debugger on errors** check box, if you want.
- Enable the **Debugger event logging** check box, if you want.
- In the **Animate settings** area, enable the desired '**Run to**' option, and specify the desired delay (in seconds).

Debug, a menu item for the Windows shell, appears on the context menu for the desktop icon of any macro.

To specify settings for editing macros

1 Click **Tools** ▸ **Settings**, and then click the **Edit** tab.

2 Specify the path and filename of the macro editor that you want to use.

3 Specify a file format that is compatible with the macro editor that you've chosen.

You must specify a macro editor if you want to edit macros.

Some macro statements are too lengthy to fit into a single line of macro code. If your macro editor automatically inserts a hard return at the end of every line, you must insert an underscore character (_) at the end of each line that wraps.

Although you can use any ASCII-based text editor to edit macros, some editors offer special features. For example,

- When you use Notepad, you can create a macro just by specifying its filename.
- When you use WordPerfect, you can use the PerfectScript Command Inserter (on the **Macro** toolbar) to insert macro commands into your macros or to edit existing macro commands.

To specify settings for playing macros

1 Click **Tools** ▸ **Settings**, and then click the **Play** tab.

2 Do any of the following:
- Specify a value in the **Play repeat count** box.
- Enable the **Security for JavaScript®** check box, if you want.
- Enable the **Show elapsed time** check box, if you want.

To specify settings for recording macros

1 Click **Tools** ▸ **Settings**, and then click the **Record** tab.

2 Do any of the following:
- From the **Script language** list box, choose the script language that you want to use.
- From the **File format** list box, choose the file format in which you want to save macros. (This file format must be compatible with the macro editor that you specify on the **Edit** page.)
- From the **Parameters per line** list box, choose **One** or **Multiple** to set the number of parameters in each macro line.
- In the **Maximum line length** box, specify a maximum line length for macro text.
- Enable the **Named parameters required** check box if you want to use only named parameters in your macros.
- Enable the **Record product prefixes on all commands** if you want to insert product prefixes in your macros.

To specify settings for the PerfectScript toolbar

1 Click **Tools** ▸ **Settings**, and then click the **Toolbar** tab.

2 Do any of the following:
- Enable the **Use large icons on toolbar buttons** check box, if you want.
- Enable the **Show text on toolbar buttons** check box, if you want.
- Assign macros to the available toolbar buttons, if you want. The macro buttons on the PerfectScript toolbar can be configured to play any desired macro.

The PerfectScript toolbar appears as a flat toolbar, similarly to other toolbars in WordPerfect Office.

Creating macros

Now that you know how to use the PerfectScript utility, you are ready to create PerfectScript macros.

This section contains the following topics:

- Migrating legacy macros
- Recording macros
- Writing and editing macros
- Compiling macros
- Playing macros
- Making macros user-friendly

Migrating legacy macros

You can use the PerfectScript utility to migrate PerfectScript macros from previous versions of WordPerfect Office to a later version of the software.

With each new version of WordPerfect Office, some commands are added, some commands are changed, and some commands become obsolete and are removed altogether. Because of such changes, macros from earlier versions of WordPerfect Office might need minor corrections when migrated to a later version of the software.

This section contains the following procedures:

- To migrate a legacy macro

Converting non-PerfectScript macros to PerfectScript format

You may also be able to use the migration process to convert a non-PerfectScript macro to PerfectScript format. (For example, if you record a macro in a non-PerfectScript language — JavaScript, Microsoft® Visual Basic®, Corel SCRIPT™, or Borland® Delphi®, as explained in “To specify settings for recording macros” on page 87 — you may subsequently want to convert that macro to PerfectScript format.) However, you must be sure to take certain precautions when converting macros.

First, be sure to review each variable, label, procedure, and function name in your existing macro, and change any names that have become reserved words. Each new version of WordPerfect Office adds PerfectScript keywords, which are reserved words that cannot be used as variable or label names in macros; in addition, the names of all macro commands are considered reserved words because they cannot be used as variable or label names. For more information on reserved words, please see the "Reserved words" topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Next, be sure to review any arrays in your existing macro (see "Working with arrays" on page 20). Please note the following:

- You can pass array slices to repeating parameter groups of commands. In early versions of PerfectScript, you could pass only entire arrays. Empty slices can now be specified, if the end index is less than the first index.
- Negative array indexes can be used in array slices. If negative, the index is considered to be end-relative, not start-relative. An index of `[-1]` represents the last element, `[-2]` represents the second-to-last element, and so on.
- You can assign non-arrays to arrays, and arrays to non-arrays. If an array is assigned to a non-array variable, the assignment is actually made to the array variable with the same name as the non-array variable, and the non-array variable is left untouched. If a non-array is assigned to an array variable, the non-array value is converted into a single element array with that value, and it is assigned to the array variable.
- Non-array values and arrays can be combined by using the operators ^^ or \. However, undefined array elements will be ignored and skipped over.

Non-array values and arrays can also be combined by using the following operators (although undefined array elements will be ignored and skipped over):

- unary operators — for information, see "Understanding operators" on page 25.
- arithmetic operators — for information, see "Understanding assignment operators" on page 27.
- relational (comparison) operators — for information, see "Understanding relational operators" on page 28.
- logical operators — for information, see "Understanding logical operators" on page 31.
- bitwise operators — for information, see "Understanding bitwise operators" on page 35.
- JavaScript operators — for information on specifying JavaScript as the macro-recording language, see "To specify settings for recording macros" on page 87.

Finally, when converting a native QuattroPro macro to PerfectScript format, you may want to know how to manually convert the syntax of each Quattro Pro command. For more information, see "Understanding the native Quattro Pro macro language" in the Quattro Pro Command Reference section of the PerfectScript Help file (**psh.chm**).

To migrate a legacy macro

1. In the PerfectScript utility for the later version of WordPerfect Office, click **File** ▸ **Play**.
2. Select the legacy macro that you want to migrate.
3. Specify a path and filename for the migrated macro, and then click **Play**.

   The macro is compiled for use with the later version of WordPerfect Office.
4. Make note of any errors that are encountered by the macro compiler, and make the necessary fixes to the macro.

   For tips on resolving macro-compilation errors, see "Troubleshooting macro-compilation errors" on page 97.

Recording macros

You can create a basic PerfectScript macro by using the PerfectScript utility to record keyboard actions in WordPerfect, Quattro Pro, or Presentations. Keyboard actions are actions that you perform by using the keyboard — for example, typing text or saving a file.

You cannot record mouse actions. However, you can use the keyboard to position the cursor by pressing an arrow key or a navigation shortcut key.

You cannot record some actions at all. However, you may be able to manually code such actions by using a macro editor (see “Writing and editing macros” on page 93).

You can also record macros from directly within WordPerfect, Quattro Pro, or Presentations. For information, please see the Help file for the application.

You can also record template macros and QuickMacros from directly within WordPerfect. For information, please see the Help file for WordPerfect.

When you record a PerfectScript macro, you record the results of your actions rather than your actual actions. For example, if you record a macro that changes the top margin of a page to **2 inches** in a WordPerfect document, PerfectScript records the WordPerfect product command `MarginTop(MarginWidth:2.0")` rather than the step-by-step method that you used to change the margin. The correct PerfectScript syntax (see “Understanding macro commands” on page 47) is automatically applied to all recorded product commands; for this reason, recording a macro helps you avoid typos and similar errors that can occur when manually coding macros.

Only product commands can be recorded. If you want to include programming commands (or complex functions such as assignments or loops) in a macro, you must manually code them. For information, see “Writing and editing macros” on page 93.

This section contains the following procedures:

- To record a macro

To record a macro

1 Open the desired WordPerfect Office application.

2 In the PerfectScript utility, click **File** ▸ **Record**.

3 Type a name for the macro, and then click **Record**.

4 Switch to the WordPerfect Office application, and then perform the keyboard actions that you want to record.

Although you cannot record mouse actions, you can use the keyboard to position the cursor by pressing an arrow key or a navigation shortcut key.

5 When you have finished recording the macro, click **File ▸ Stop** in the PerfectScript utility.

When you record a macro, the PerfectScript utility automatically records the `Application` command for the appropriate WordPerfect Office application. This command indicates the application to which the macro belongs.

Some actions cannot be recorded. However, you may be able to manually code these actions by using a macro editor (see "Writing and editing macros" on page 93).

If you need to stop recording temporarily (for example, to locate a feature or to experiment with the effect of a feature before you record the command), click **File ▸ Pause** in the PerfectScript utility. Click **Pause** again to resume recording the macro.

You can specify the settings to use when recording macros. For information, see "To specify settings for recording macros" on page 87.

Writing and editing macros

If you prefer to write (rather than record) a PerfectScript macro, you can do so by using a macro editor — or even by typing in a blank document. Manually coding a PerfectScript macro requires an understanding of both the PerfectScript language and the principles of computer programming.

A macro editor, as its name implies, lets you edit the code for an existing macro. You can edit a macro if you want to change how that macro operates.

If you choose WordPerfect as your macro editor, you can easily insert PerfectScript macro commands into your macro code by using the Command Inserter feature of the Command Browser. The Command Inserter feature lets you choose macro commands and parameters from the provided lists and then insert the resulting syntax into your macro code. Using the Command Inserter saves you time and helps you avoid typos and similar errors that can occur when manually coding macros.

If you choose WordPerfect as your macro editor, you must disable the SmartQuotes feature. To disable SmartQuotes in WordPerfect, click **Tools** ▸ **QuickCorrect**™, click the **SmartQuotes** tab, and then disable the **Use double quotation marks as you type** check box.

If you do not choose WordPerfect as your macro editor, you must be sure to apply the correct syntax to all the macro commands that you type. For information about macro-command syntax, see “Understanding macro commands” on page 47.

This section contains the following procedures:

- To write a macro
- To edit a macro
- To insert a macro command into macro code

Formatting macros

If you want to improve the readability of a macro, you can format it to include tabs, spaces, and even font styles or other text-appearance changes. Formatting a macro does not affect how it works.

For example, WordPerfect records the following macro in this default format:

```
PosDocBottom()
Type("Sincerely")
HardReturn()
HardReturn()
HardReturn()
HardReturn()
Type("Ms. Sharon Openshaw")
HardReturn()
Type("Vice President, Marketing")
```

However, you can type spaces between components and blank lines between tasks, as follows:

```
PosDocBottom()


Type ("Sincerely")


HardReturn()
HardReturn()
HardReturn()
HardReturn()

Type ("Ms. Sharon Openshaw")

HardReturn()
Type ("Vice President, Marketing")
```

To write a macro

1 Type the macro code in a macro editor or in a blank document.

2 Save the macro as a file with a **.wcm** extension.

You can specify the settings to use when writing macros in a macro editor (see “To specify settings for editing macros” on page 86).

To edit a macro

1 In the PerfectScript utility, click **File ▸ Edit**.

2 Select the macro that you want to edit, and then click **Edit**.
If necessary, click **Convert** to convert the macro for editing.

3 Make the desired changes to the macro, and close the macro editor.

In Windows, you can access the **Edit** command for a macro by right-clicking that macro.

You can specify the settings to use when editing macros (see “To specify settings for editing macros” on page 86).

To insert a macro command into macro code

1 In the macro code, click where you want to insert a macro command.

2 In the PerfectScript utility, click **Help ▸ Macro Command Browser**.

3 From the **Command type** list box, choose the type of command that you want to insert.

4 In the **Commands** list, double-click the command that you want to insert.

5 In the **Parameters** list, double-click the parameter that you want to insert.

If the parameter has enumerations, double-click the desired enumeration to insert both the parameter and its enumeration.

6 Repeat step 5 for each additional parameter.

7 Click **Return values** if you want to return value enumerations for commands that have enumerations as return values.

8 If desired, manually edit the command in the **Command edit** list box.

9 Click **Insert** to copy the command from the **Command edit** box and insert it into the macro code.

You can use the Command Inserter feature only when using WordPerfect as your macro editor. For information about choosing a macro editor, see "To specify settings for editing macros" on page 86.

For most macro commands, a default value is passed when an optional enumeration parameter is omitted — and in some cases, omitting an optional enumeration parameter performs a different function altogether. In the Command Browser, any default enumeration values for a parameter are displayed in bold text in the **Parameters** list.

For some macro commands, the default parameter value is a combination of enumerations. In this scenario, several enumerations may be defined as synonyms that have the same value; in the Command Browser, any such enumerations are highlighted.

You can display Help for any macro command in the Command Browser by right-clicking that command in the **Commands** list.

Compiling macros

To create a functioning macro from macro code, you must use a "compiler." PerfectScript macros are automatically compiled when they are recorded or played, but they can be manually compiled at any time by using the PerfectScript utility.

You can also compile macros from directly within WordPerfect or Presentations. For information, please see the Help file for the application.

This section contains the following procedures:

- To compile a macro

Understanding compilers

In machine (computer) language, every word is a binary numeral that consists of zeros and ones. Consider the following examples:

- In binary notation, the first three letters of the alphabet are 1000001, 1000010, and 1000011.
- The binary result of 4 + 5 is 1001.

Working with binary numerals can be awkward, so English-based programming languages (such as Basic, Pascal, and C) were designed to simplify the process of writing programs. A programming language is used to write program code in an editor or word processor, and that program code is then saved as a source file. However, computers can execute only object files, not source files. For this reason, a program compiler is required to create an object file by making a copy of the source file and translating that copy into machine language.

Macro languages are similar to programming languages. A macro language (such as PerfectScript) is used to write macro code in an editor or word processor, and that macro code is then saved as a source file. However, rather than create a separate object file from the source file, a macro compiler creates an object and saves it in a hidden area of the source file. This hidden object is destroyed when the source file is edited and regenerated when the source file is recompiled.

A macro is therefore a compiled source file that contains instructions that are executed when that macro is played. The PerfectScript macro compiler is used to compile or "translate" PerfectScript macro code into a usable format for WordPerfect Office applications.

Troubleshooting macro-compilation errors

The PerfectScript macro compiler is useful for troubleshooting problems with your PerfectScript macros. When the compiler locates an error, it displays a dialog box that contains general information about the problem. However, the compiler can make only a best guess as to what a macro is intended to accomplish; as a result, the compiler may direct you to a problem rather than specifically identify that problem.

If you receive an error message while compiling a macro, you can continue the macro-compilation process if you want to check for additional errors, or you can cancel the macro-compilation process altogether. In either case, the macro cannot be played until you correct all errors and successfully compile the macro.

The following syntax errors can cause a macro-compilation error:

- A command name is misspelled
- A semicolon is missing between macro-command parameters
- A comma, instead of a semicolon, is used between macro-command parameters
- A parenthesis is missing
- A (double) quotation mark is missing
- A (double) quotation mark is inserted by using the SmartQuotes feature in WordPerfect
- A macro command is missing from a conditional statement
- A macro command is missing from a loop statement
- A calling statement is undefined

In addition, the following conditions can cause a macro-compilation error:

- No "return" statement is found in the body of a user-defined function. In this case, a `return (0)` statement is generated.
- A "return" statement with no return value is found in the body of a user-defined function. In this case, a value of `0` is returned.
- The macro appears to be empty when compiled. This issue can occur when a previously compiled macro has its source removed; compiling such a macro destroys the existing (compiled) macro object.
- An obsolete or unsupported feature is found in a macro during compilation. These warnings can be safely ignored to produce a successful macro — they serve as reminders only. Warnings are displayed when an old EN English synonym is used in the `Application` statement (US, UK, CE, OZ), or when an obsolete or unsupported command, enumeration, or parameter is used.

When correcting macro-compilation errors, work through the macro code from beginning to end, and focus on the errors for which the solution seems most apparent. Leave the errors with less apparent solutions until later — some of these errors may be corrected by resolving the more obvious errors.

To compile a macro

1 In the PerfectScript utility, click **File ▸ Compile**.

2 Select the macro that you want to compile, and then click **Compile**.

You can also

Cancel the compilation of a macro	In the **Compile progress** dialog box, do one of the following: • Click the **Cancel** button. • Press **Enter**.

If you want to compile a legacy macro or a non-PerfectScript macro, you must first convert it to the current PerfectScript format. For information, see “Migrating legacy macros” on page 89.

When a macro is compiled, warnings are displayed for any labels or routines that are not defined by that macro. However, the compiled macro will function correctly if it calls another macro that defines those labels and routines.

In Windows, you can access the **Compile** command for a macro by right-clicking that macro.

You can specify the settings to use when compiling macros. For information, see “To specify settings for compiling macros” on page 85.

Playing macros

You can perform the operations that are specified in a PerfectScript macro by using the PerfectScript utility to play that macro.

When you play a PerfectScript macro, the PerfectScript utility determines which application is associated with that macro. The PerfectScript utility then checks the registry for the path to the EXE file for the application. If that path is not in the registry, you are prompted to specify the location of that EXE file.

You can also play macros from directly within WordPerfect, Quattro Pro, or Presentations. For information, please see the Help file for the application.

You can also play template macros and QuickMacros from directly within WordPerfect. For information, please see the Help file for WordPerfect.

This section contains the following procedures:

- To play a macro

Troubleshooting macro run-time errors

A run-time error is a problem that occurs while a macro is playing.

When a run-time error is encountered, an error message displays the location of the problem in the macro code. For this reason, the best way to troubleshoot a run-time error is to consult its error message.

To play a macro

1 In the PerfectScript utility, click **File** ▸ **Play**.

2 Select the macro that you want to play, and then click **Play**.

You can also

Pause (or resume) a macro	In the PerfectScript utility, click **File** ▸ **Pause**.
Stop a macro	In the PerfectScript utility, click **File** ▸ **Stop**.

Every time a macro is played, it is recompiled and then saved. If you do not want to save over a macro when you play it, you can create a new version of it by specifying a different path or filename in the **Play macro** dialog box.

In Windows, you can access the commands for playing, pausing, and stopping a macro by right-clicking that macro.

You can specify the settings to use when playing macros. For information, see "To specify settings for playing macros" on page 86.

Making macros user-friendly

From directly within WordPerfect, Quattro Pro, or Presentations, you can assign a macro to a keystroke, menu, toolbar, or property bar. For information, see the Help file for the application.

If you want to make a macro even more user-friendly, you can create a dialog box for it by using the Dialog Editor feature of the PerfectScript utility. For information, see "Creating dialog boxes for macros" on page 101.

Creating dialog boxes for macros

You can create dialog boxes for your macros if you want to provide an interface between the application and the user.

This section contains the following topics:

- Understanding dialog boxes
- Setting up dialog boxes for macros
- Setting up controls for dialog boxes
- Setting up callbacks for dialog boxes
- Testing dialog boxes
- Displaying dialog boxes

Understanding dialog boxes

Dialog boxes provide an interface between the application and the user.

Dialog boxes come in two types:

- *modal* — A modal dialog box locks the application until the user acts on that dialog box and closes it. The **File | Open** dialog box in WordPerfect is an example of a modal dialog box because focus remains on this dialog box until it is released.
- *modeless* — A modeless dialog box does not lock the application, so the user can move between the dialog box and the application as necessary. The **Find and Replace** dialog box in WordPerfect is an example of a modeless dialog box because you can continue working in a document while this dialog box is displayed.

Of the two dialog-box types, modal is the more common.

To use a modeless dialog box in your macro, you must enable the `Display` property and disable the `InhibitInput` property.

As a macro programmer, you can take advantage of dialog boxes in PerfectScript if you want to obtain information or data from a user. Dialog boxes that are created in a macro can be used to ask questions or otherwise gather data — data that, in turn, can be used to determine the flow and control of the macro.

You can create a dialog box in one of two ways:

- **by writing PerfectScript code** — The PerfectScript language provides programming commands for manually coding dialog boxes.
- **by using the PerfectScript Dialog Editor** — The PerfectScript Dialog Editor provides a graphical development environment for designing dialog boxes quickly and easily.

The Dialog Editor does not let you edit macros, only to define dialog boxes for them.

The Dialog Editor works only with macros that are in WordPerfect format. For this reason, you can open the Dialog Editor either from within the PerfectScript utility or from within WordPerfect, as explained in "To display the Dialog Editor" on page 84.

The first step in creating a dialog box is explained in "Setting up dialog boxes for macros" on page 102.

The second step in creating a dialog box is explained in "Setting up controls for dialog boxes" on page 108.

The third step in creating a dialog box is explained in "Setting up callbacks for dialog boxes" on page 124.

The fourth step in creating a dialog box is explained in "Testing dialog boxes" on page 131.

The fifth and final step in creating a dialog box is explained in "Displaying dialog boxes" on page 131.

Setting up dialog boxes for macros

You can set up a dialog box for a macro, either by writing PerfectScript code or by using the PerfectScript Dialog Editor.

Setting up dialog boxes by using PerfectScript code

You can use the PerfectScript programming command `DialogDefine` to set up a dialog box.

The following code provides an example of a dialog box that is set up by using the `DialogDefine` command.

```
DialogDefine(Dialog: "MainDialog"; Left: 50; Top: 50; Width: 150; _
Height: 100; Style: OK! | Percent!; Caption: "Example Dialog Box")
```

Remember that because parameter names are optional, the following command is equivalent to the preceding one:

```
DialogDefine (1000; 50; 50; 200; 125; OK! | Percent!; _
"Example Dialog Box")
```

You can use + or | in parameters where multiple values can be specified. In the preceding example, the syntax `OK! | Percent!` applies both the `OK!` enumeration and the `Percent!` enumeration to the `Style` parameter.

The following table describes the purpose of each parameter in the preceding example.

Parameter (or parameters)	**Description**
`Dialog`	Specifies the name of the dialog box, which is used to refer to the dialog box throughout the macro code. This name can consist of letters or numbers (or both), and it must be unique. In the preceding example, the name `MainDialog` is assigned to the dialog box.
`Left` and `Top`	Work together to specify the position of the top-left corner of the dialog box. Dialog boxes are positioned in dialog-box units, whereby a vertical unit equals 1/8 the font height and a horizontal unit equals 1/4 the font width. In the preceding example, the `Left` and `Top` parameters are assigned a value of `50` and are used with the `Percent!` enumeration of the `Style` parameter. As a result, the dialog box is centered on the screen.

Parameter (or parameters)	Description
`Width` and `Height`	Work together to determine the size of the dialog box. Dialog boxes are sized in dialog-box units, whereby a vertical unit equals 1/8 the font height and a horizontal unit equals 1/4 the font width. In the preceding example, the `Width` and `Height` parameters are assigned values of `150` and `100` (respectively).
`Style`	Specifies one or more dialog-box styles. These styles are used to determine the appearance and function of the dialog box. In the preceding example, the `Style` parameter is assigned the enumerations `OK!` and `Percent!`. The `OK!` enumeration adds an **OK** button to the dialog box. The `Percent!` enumeration sets the `Left` and `Top` parameters to use the percentage of the screen width or height minus the width or height of the dialog box.
`Caption`	Specifies the text to be displayed in the caption (title) bar In the preceding example, the text `"Example Dialog Box"` is assigned to the caption bar.

For more information about the `DialogDefine` command, please see the DialogDefine topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Setting up dialog boxes by using the Dialog Editor

You can use the Dialog Editor to add a dialog box to a macro. For information, see “To add a dialog box to a macro by using the Dialog Editor” on page 105.

You can also use the Dialog Editor to set the properties for a dialog box. You can use properties to specify the location and size of a dialog box, as well as its caption, Help file, Help key, type (modal or modeless), frame type, and attributes. For more information, see “To set the properties for a dialog box by using the Dialog Editor” on page 107.

You can use the Dialog Editor to choose the typeface and point size for the text in a dialog box. Changes in font size and style affect the size of the dialog box. For more information, see “To set the font for a dialog box by using the Dialog Editor” on page 107.

Finally, you can use the Dialog Editor to save a dialog box in the current macro. For information, see “To save a dialog box in the current macro by using the Dialog Editor” on page 108.

To add a dialog box to a macro by using the Dialog Editor

1 In the PerfectScript utility, click **Tools ▸ Dialog Editor**.

The **Edit Macro Dialogs** dialog box appears.

2 Select the desired macro, and then click **OK**.

The Dialog Editor for that macro appears.

3 Click **File ▸ New**, and then type a name for the new dialog box.

4 Set the properties for the dialog box. For information, see “To set the properties for a dialog box by using the Dialog Editor” on page 107.

5 Select the dialog box, and then click **File ▸ Open**.

The dialog box is opened for editing in the Dialog Editor.

6 Do any of the following:

- Set the font for the dialog box. For information, see “To set the font for a dialog box by using the Dialog Editor” on page 107.
- Add controls to the dialog box. For information, see “Setting up controls for dialog boxes” on page 108.

7 Click **File ▸ Save**, and then click **File ▸ Close**. For more information, see “To save a dialog box in the current macro by using the Dialog Editor” on page 108.

8 Test the dialog box. For information, see “Testing dialog boxes” on page 131.

9 Write macro code for opening and closing the dialog box. For information, see “Displaying dialog boxes” on page 131.

Dialog-box names are case-sensitive. Be sure to correctly reference them in your macros.

When you save a dialog box, its size and position are recorded. When you open a saved dialog box, its size and position are loaded, applied, and maintained during the current session of PerfectScript.

You can also

Copy a dialog box by using the Dialog Editor	In the PerfectScript utility, click **Tools** ▶ **Dialog Editor**. Select the macro that contains the dialog box, and then click **OK**. Click **Edit** ▶ **Copy**. Open the macro file to which you want to copy the dialog box, and then click **Edit** ▶ **Paste**. Rename the pasted dialog box, if desired.
Edit a dialog box by using the Dialog Editor	In the PerfectScript utility, click **Tools** ▶ **Dialog Editor**. Select the macro that contains the dialog box, and then click **OK**. Double-click the dialog box that you want to edit, and then edit the dialog box as desired. Click **Edit** ▶ **Save** to save the edited dialog box in the current macro file.
Rename a dialog box by using the Dialog Editor	In the PerfectScript utility, click **Tools** ▶ **Dialog Editor**. Select the macro that contains the dialog box, and then click **OK**. Select the dialog box, and then click **File** ▶ **Rename**. Type a new name for the dialog box. **NOTES:** • Dialog-box names are case-sensitive. Be sure to correctly reference them in your macros. • Renaming a dialog box gives it a new name in the current macro file, but it does not change the name that is displayed on the caption bar. • Remember to update the macro code with new name of the dialog box.

You can also

Delete a dialog box by using the Dialog Editor	In the PerfectScript utility, click **Tools ▸ Dialog Editor**. Select the macro that contains the dialog box, and then click **OK**. Select the dialog box, and then click **File ▸ Delete**. Click **Yes** to confirm that you want to delete the dialog box. **NOTE:** Remember to remove all references to the deleted dialog box from the macro code.

To set the properties for a dialog box by using the Dialog Editor

1 In the PerfectScript utility, click **Tools ▸ Dialog Editor**.

2 Select the macro that contains the dialog box, and then click **OK**.

3 Select the dialog box, and then click **File ▸ Properties**.

4 In the **Dialog Properties** dialog box that appears, specify any of the following properties for the dialog box:
- location and size
- caption
- Help file
- Help key
- dialog-box type
- frame type
- attributes

You can choose between two dialog-box types: modal and modeless. A modal dialog box locks the macro until the user acts on that dialog box and closes it. A modeless dialog box does not lock the macro, so the user can move between the dialog box and the macro as necessary.

To set the font for a dialog box by using the Dialog Editor

1 In the PerfectScript utility, click **Tools ▸ Dialog Editor**.

2 Select the macro that contains the dialog box, and then click **OK**.

3 Select the dialog box, and then click **File ▸ Open**.

4 Click **Dialog ▸ Font**.

5 In the **Dialog Font** dialog box that appears, specify any of the following font attributes:
   - style
   - size

The font size that you choose affects the size of the dialog box — the larger the font, the larger the dialog box.

The caption font remains constant for all dialog boxes.

To save a dialog box in the current macro by using the Dialog Editor

1 With the dialog box open for editing in the Dialog Editor, click **File** ▸ **Save** to save the dialog box in the current macro file.

2 Click **File** ▸ **Close** to stop editing the dialog box and close its Dialog Editor.

Setting up controls for dialog boxes

You can set up controls for a dialog box, either by writing PerfectScript code or by using the PerfectScript Dialog Editor. Controls are input or output windows where the user interacts with a dialog box and its parent application. You can use any of the following control types in your dialog boxes:

- **bitmap** — see "Using bitmaps in dialog boxes" on page 110
- **button** — see "Using buttons in dialog boxes" on page 111
- **check box** — see "Using check boxes in dialog boxes" on page 111
- **color wheel** — see "Using color wheels in dialog boxes" on page 112
- **combo box** — see "Using combo boxes in dialog boxes" on page 112
- **counter** — see "Using counters in dialog boxes" on page 113
- **custom control** — see "Using custom controls in dialog boxes" on page 113
- **date control** — see "Using date controls in dialog boxes" on page 113
- **edit box** — see "Using edit boxes in dialog boxes" on page 114
- **filename box** — see "Using filename boxes in dialog boxes" on page 115
- **frame** — see "Using frames in dialog boxes" on page 115
- **group box** — see "Using group boxes in dialog boxes" on page 115
- **line** — see "Using lines in dialog boxes" on page 116
- **list control** — see "Using list controls in dialog boxes" on page 116
- **progress indicator** — see "Using progress indicators in dialog boxes" on page 117

- **scroll bar** — see “Using scroll bars in dialog boxes” on page 117
- **static text** — see “Using static text in dialog boxes” on page 118
- **viewer** — see “Using viewers in dialog boxes” on page 118

You can add controls to a dialog box by writing macro code. The PerfectScript language provides a programming command for creating each control type — simply insert these commands after the `DialogDefine` command (see “Setting up dialog boxes by using PerfectScript code” on page 102) for the dialog box.

In the following example, the PerfectScript programming command `DialogAddText` is used to add a static-text control (named `Control1`) to the dialog box named `MainDialog` by using the specified parameters:

```
DialogAddText (Dialog: "MainDialog"; Control: "Control1"; _
Left: 10; Top: 10; Width: 50; Height: 9; Style: Left!; _
Text: "Edit control:")
```

In the following example, the PerfectScript programming command `DialogAddEditBox` is used to add an edit-box control (named `Control2`) to the dialog box named `MainDialog` by using the specified parameters:

```
DialogAddEditBox (Dialog: "MainDialog"; Control: "Control2"; _
Left: 10; Top: 25; Width: 125; Height: 25; Style: Left! | _
VScroll! | Multiline!; vReturn; 1000)
```

For information about the PerfectScript programming commands for creating dialog-box controls, please see the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Alternatively, you can use the Dialog Editor to set up controls in any of the following ways:

- by adding controls to a dialog box. For information, see “To add a control to a dialog box by using the Dialog Editor” on page 119.
- by setting the properties for the controls in a dialog box. The properties that are available to a control depend on the control type. For information, see “To set the properties for a control by using the Dialog Editor” on page 120.
- by positioning the controls in a dialog box, either at specified locations or in relation to each other. For information, see “To position one or more controls by using the Dialog Editor” on page 122.
- by assigning behaviors to the controls in a dialog box. For information, see “To assign behaviors to one or more controls by using the Dialog Editor” on page 124.

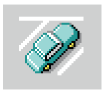
Some controls require the use of a hot spot — an invisible control that closes the dialog box when the user clicks the defined area. (The response for a hot spot can be redefined with a callback function.) The PerfectScript command for creating a hot spot is `DialogAddHotSpot`. For information about this command, please see the DialogAddHotSpot topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

The PerfectScript language lets you create one type of control that is not supported by the Dialog Editor: an icon control, which is represented by the programming command `DialogAddIcon`. An icon control does not accept input, unless it is used in a callback function with the PerfectScript command `DialogAddHotSpot`. For information about the `DialogAddIcon` and `DialogAddHotSpot` commands, please see the DialogAddIcon and DialogAddHotSpot topics in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Variables that are associated with controls work the same way with dialog boxes that are created by using PerfectScript code as they do with dialog boxes that are created by using the Dialog Editor. If a variable exists, its value is set into the controls when the dialog box is opened, and the value in the controls is set into the variables when the dialog box is closed. For more information about using PerfectScript code to open and close dialog boxes, see “Displaying dialog boxes” on page 131.

Using bitmaps in dialog boxes

You can display a bitmap as a control. By default, the bitmap appears without a border on the background of the dialog box; however, for visual clarity, you can give the bitmap an outline. A bitmap control looks like this:



The PerfectScript command for creating a bitmap control is `DialogAddBitmap`. For information about this command, please see the DialogAddBitmap topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

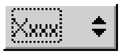
A bitmap control does not accept input, unless it is used in a callback function with the PerfectScript command `DialogAddHotSpot`. For information about this command, please see the DialogAddHotSpot topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Using buttons in dialog boxes

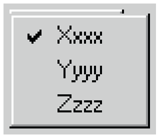
You can add three kinds of buttons to a dialog box: pop-up buttons, push buttons, and radio buttons.

A pop-up button displays a list of options when clicked. The button itself shows the selected option.

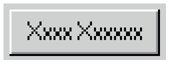
A pop-up button looks like this when it is closed:



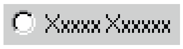
A pop-up button looks like this when it is clicked:



A push button activates a specific action — such as **OK**, **Cancel**, or **Help** — when clicked. A push button looks like this:



Radio buttons are used in groups to represent options that are mutually exclusive. Enabling one radio button disables another. A radio button looks like this:

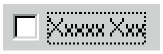


The PerfectScript command for creating a push-button control is `DialogAddPushButton`, and the command for creating a radio-button control is `DialogAddRadioButton`. For information about these commands, please see the DialogAddPushButton and DialogAddRadioButton topics in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

A radio-button control uses a callback function to activate user-defined responses.

Using check boxes in dialog boxes

Check boxes are used in groups to represent options that are compatible. Clicking an empty check box enables that option, while clicking a marked check box disables that option. A check box looks like this:



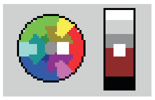
The PerfectScript command for creating a check-box control is `DialogAddCheckBox`. For information about this command, please see the DialogAddCheckBox topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

A check-box control uses a callback function to activate user-defined responses.

You can define a check box as "Three State" if you want it to provide a state of unavailability (that is, a state in which the box is checked and grayed) in addition to an enabled state and a disabled state.

Using color wheels in dialog boxes

A color wheel lets the user select a color based on values of hue, lightness, and saturation. A color wheel looks like this:



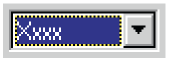
The PerfectScript command for creating a color-wheel control is `DialogAddColorWheel`. For information about this command, please see the DialogAddColorWheel topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

In a color-wheel control, you can use the arrow keys to move the color selection. Hold down **Ctrl** while using the arrow keys to change the value of the color-saturation bar.

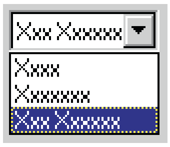
Using combo boxes in dialog boxes

A combo box combines an edit box (see "Using edit boxes in dialog boxes" on page 114) with a list box (see "Using list controls in dialog boxes" on page 116). You can enter text in the edit box, or you can double-click a list item to insert it.

A combo box looks like this when it is closed:



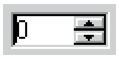
A combo box looks like this when it is clicked:



The PerfectScript command for creating a combo-box control is `DialogAddComboBox`. For information about this command, please see the DialogAddComboBox topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Using counters in dialog boxes

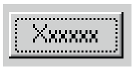
A counter lets the user enter numeric data in an edit box (see "Using edit boxes in dialog boxes" on page 114), either by typing in the edit box or by clicking the built-in incrementor/decrementor control. A counter looks like this:



The PerfectScript command for creating a counter control is `DialogAddCounter`. For information about this command, please see the DialogAddCounter topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Using custom controls in dialog boxes

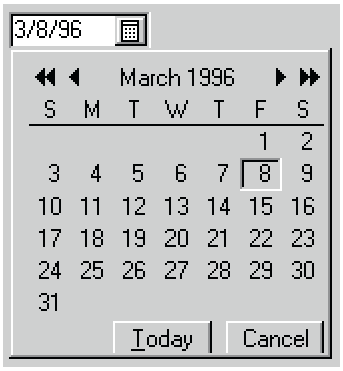
A custom control lets you, the macro programmer, create a control by specifying its settings for text, class, and attributes. A custom control looks like this:



The PerfectScript command for creating a custom control is `DialogAddControl`. For information about this command, please see the DialogAddControl topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Using date controls in dialog boxes

A date control displays an edit box (see "Using edit boxes in dialog boxes" on page 114) and a calendar icon. The user can enter a date by typing in the edit box or by clicking the calendar icon and choosing a date from the calendar that appears. A date control looks like this when the calendar icon is clicked:



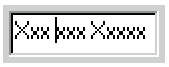
The PerfectScript command for creating a date control is `DialogAddDate`. For information about this command, please see the DialogAddDate topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

In a date control, you can use the following keyboard shortcuts to change the date more quickly:

- holding **Ctrl** while clicking an arrow icon — increases the tens column
- holding **Alt** while clicking an arrow icon — increases the hundreds column
- pressing **Alt** + arrow key — changes the month by one month
- pressing **Page Up** or **Page Down** — changes the year
- pressing **Alt** — changes the year by one year
- pressing **Alt** + **Ctrl** — changes the year by 10 years
- pressing **Alt** + **Shift** — changes the year by 100 years

Using edit boxes in dialog boxes

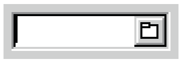
An edit box lets the user type text, or it lets the macro type text on the user's behalf. An edit box can have one or more lines. An edit box looks like this:



The PerfectScript command for creating an edit-box control is `DialogAddEditBox`. For information about this command, please see the DialogAddEditBox topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Using filename boxes in dialog boxes

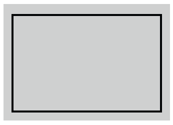
A filename box displays an edit box (see "Using edit boxes in dialog boxes" on page 114) and a folder button. The user can specify a file either by typing the filename (and its path) or by clicking the folder button to display a **Browse** dialog box. A filename box looks like this:



The PerfectScript command for creating a filename-box control is `DialogAddFileNameBox`. For information about this command, please see the DialogAddFileNameBox topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Using frames in dialog boxes

A frame can be used to visually group the items in a dialog box, or to act as a design element. A frame looks like this:

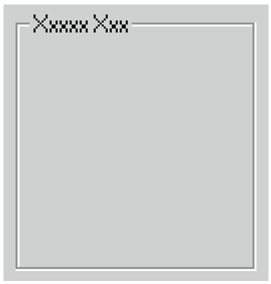


The PerfectScript command for creating a frame control is `DialogAddFrame`. For information about this command, please see the DialogAddFrame topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

A frame control does not accept input.

Using group boxes in dialog boxes

A group box visually groups controls by using a titled frame. A group box looks like this:



The PerfectScript command for creating a group-box control is `DialogAddGroupBox`. For information about this command, please see the DialogAddGroupBox topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

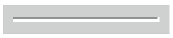
A group-box control does not accept input.

A group box groups controls visually, but not functionally. To make the controls in a group box function as a group, you can use the **Control Order** and **Control Groups** features in the Dialog Editor, as explained in "To assign behaviors to one or more controls by using the Dialog Editor" on page 124.

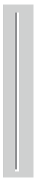
Using lines in dialog boxes

You can use a horizontal line or a vertical line to visually separate the items in a dialog box.

A horizontal line looks like this:



A vertical line looks like this:



The PerfectScript command for creating a horizontal-line control is `DialogAddHLine`, while the command for creating a vertical-line control is `DialogAddVLine`. For information about these commands, please see the DialogAddHLine and DialogAddVLine topics in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

A line control does not accept input.

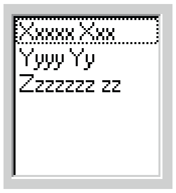
Using list controls in dialog boxes

A list control displays a series of options (or "list items") from which to choose. A list control takes one of the following forms:

- a pop-up list — presents the list items in a pop-up window
- a drop-down list — presents the list items in a window that extends outward from the list control

- a list box — presents the list items in a window that acts much like a viewer (see "Using viewers in dialog boxes" on page 118)

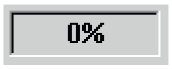
A list box looks like this:



The PerfectScript command for creating a list-box control is `DialogAddListBox`, while the command for creating a list item is `DialogAddListItem`. For information about these commands, please see the DialogAddListBox and DialogAddListItem topics in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Using progress indicators in dialog boxes

A progress indicator displays the progress of a process as it runs. A progress indicator looks like this:

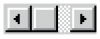


The PerfectScript command for creating a progress-indicator control is `DialogAddProgress`. For information about this command, please see the DialogAddProgress topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

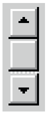
Using scroll bars in dialog boxes

A scroll bar lets the user scroll the viewable area. A horizontal scroll bar moves the viewable area left and right, while a vertical scroll bar moves the viewable area up and down.

A horizontal scroll bar looks like this:



A vertical scroll bar looks like this:

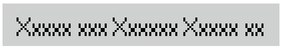


The PerfectScript command for creating a scroll-bar control is `DialogAddScrollBar`. For information about this command, please see the DialogAddScrollBar topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

A scroll-bar control can use a callback function to activate user-defined responses.

Using static text in dialog boxes

Static text provides the user with one or more lines of read-only information, such as instructions. Static text looks like this:



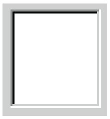
The PerfectScript command for creating a static-text control is `DialogAddText`. For information about this command, please see the DialogAddText topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

A static-text control does not accept input.

You can copy static text from most dialog boxes that are part of a macro system. However, you cannot copy static text from user-defined dialog boxes.

Using viewers in dialog boxes

A viewer displays a read-only, scrollable text file. A viewer looks like this:



The PerfectScript command for creating a viewer control is `DialogAddViewer`. For information about this command, please see the DialogAddViewer topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

To add a control to a dialog box by using the Dialog Editor

1 In the PerfectScript utility, click **Tools ▸ Dialog Editor**.

2 Select the macro that contains the dialog box, and then click **OK**.

The Dialog Editor for that macro appears.

3 Select the dialog box, and then click **File ▸ Open**.

The dialog box is opened for editing in the Dialog Editor.

4 Add the desired type of control by clicking its corresponding command on the **Control** menu (or by clicking its corresponding toolbar icon):

- bitmap — **Control ▸ Bitmap**
- button, push — **Control ▸ Buttons ▸ Push**
- button, radio — **Control ▸ Buttons ▸ Radio**
- check box — **Control ▸ Check Box**
- color wheel — **Control ▸ Color Wheel**
- combo box — **Control ▸ Combo Box**
- counter — **Control ▸ Counter**
- custom control — **Control ▸ Custom**
- date control — **Control ▸ Date**
- edit box — **Control ▸ Edit Box**
- filename box — **Control ▸ Filename Box**
- frame — **Control ▸ Frame**
- group box — **Control ▸ Group Box**
- line, horizontal — **Control ▸ Lines ▸ Horizontal**
- line, vertical — **Control ▸ Lines ▸ Vertical**
- list control — **Control ▸ List Box**
- progress indicator — **Control ▸ Progress Indicator**
- scroll bar, horizontal — **Control ▸ Scroll Bars ▸ Horizontal**
- scroll bar, vertical — **Control ▸ Scroll Bars ▸ Vertical**
- static text — **Control ▸ Static Text**
- viewer — **Control ▸ Viewer**

5 Position the pointer where you want the top-left corner of the control to appear, and then click.

You can also

Select a control in the Dialog Editor	Click the control.

You can also

Select multiple controls in the Dialog Editor	Hold down **Shift**, and click the controls. The last control that you click is called the “anchor control,” and it appears with black squares around it
Move a control in the Dialog Editor	Drag the control. **TIP:** For information about precisely positioning a control, see “To position one or more controls by using the Dialog Editor” on page 122.
Resize a control in the Dialog Editor	Drag one of the handles for the control.
Copy a control in the Dialog Editor	Right-click the control, and then click **Copy**. Drag the copy to position it. **NOTE:** The copy has all the properties of the original control, but you can change them if desired. **TIP:** You can also copy a control by holding down **Ctrl** while selecting it.
Edit a control in the Dialog Editor	Double-click the control, and then set its properties. For information, see “To set the properties for a control by using the Dialog Editor” on page 120.
Delete a control in the Dialog Editor	Do one of the following: • Right-click the control, and then click **Delete**. • Select the control, and then press **Delete**.

To set the properties for a control by using the Dialog Editor

1 With the dialog box open for editing in the Dialog Editor, display the properties for the control by doing one of the following:
   - Double-click the control.
   - Right-click the control, and then click **Properties**.

   The **Properties** dialog box for the control appears.

2 Set the properties for the control. The properties that are available to a control depend on the control type, as follows:

- bitmap — location and size; named region; variable; filename; attributes
- button, push — location and size; named region; text; type
- button, radio — location and size; named region; variable; text; type; text placement; initial state
- check box — location and size; named region; variable; text; type; text placement; initial state
- color wheel — location and size; named region; variable; initial color values
- combo box — location and size; named region; variable; style; current item list; type; attributes
- counter — location and size; named region; variable; values; attributes
- custom control — location and size; named region; text; class; styles
- date control — location and size; named region; variable; initial date; attributes
- edit box — location and size; named region; variable; style; text; type; justification; capitalization; attributes; automatic scroll; scroll bar
- filename box — location and size; named region; variable; folder; template; type
- frame — location and size; named region; type; color
- group box — location and size; named region; text
- line, horizontal — location and size; named region
- line, vertical — location and size; named region
- list control — location and size; named region; variable; style; current item list; attributes
- progress indicator — location and size; named region
- scroll bar, horizontal — location and size; named region; variable; values; alignment and sizing
- scroll bar, vertical — location and size; named region; variable; values; alignment and sizing
- static text — location and size; named region; variable; style; text; justification; prefix; type
- viewer — location and size; named region; variable; filename

A control name is specified by the **Named region** property. Control names are case-sensitive, so be sure to correctly reference them in your macros.

You can use the **Properties** dialog box to determine a precise location and a precise size for a control. However, you can roughly position a control by dragging it around the dialog box, and you can roughly size it by dragging one of its handles. For more information about positioning controls, see “To position one or more controls by using the Dialog Editor” on page 122.

You can also

Create or edit the item list for a combo box or a list control	In the **Properties** dialog box for the control, click **Create/Edit List**, and then do any of the following: • add an item — Type the name of the item in the **List item** box, and then click **Add**. The item is added to the list, in the **List** box. • select an item in the **List** box— Click the item. • edit an item — Select the item, type a new value in the **List item** box, and then click **Replace**. • move an item up the list — Select the item, and then click **Move Up** until the item reaches the desired position. • move an item down the list — Select the item, and then click **Move Down** until the item reaches the desired position. • set the default list item — Select the item, and then click **Set Initial**. The item appears in the **Initial** line. • delete an item — Select the item, and then click **Delete**. • sort the list alphabetically — Enable the Sort List check box. **NOTE:** Enabling this check box prevents you from manually rearranging the list items.

To position one or more controls by using the Dialog Editor

- With the dialog box open for editing in the Dialog Editor, position one or more controls by using any of the methods in the following table.

To	Do the following
Position a single control roughly	Drag the control to the desired position in the dialog box. **TIP:** You can roughly size the control by dragging one of its handles.

To	Do the following
Position a single control precisely	Double-click the control to open its **Properties** dialog box. In the **Location and size** area, determine the position of the top-left corner of the control by specifying values in the **Left** and **Top** boxes. **TIP:** You can precisely size the control by specifying values in the **Width** and **Height** boxes.
Position multiple controls roughly	Select the controls, and then drag them to the desired position in the dialog box.
Position multiple controls in relation to each other	Select the controls, and then position them in any of the following ways: • aligned with one side of the anchor control — Click **Align**, and then click **Left**, **Right**, **Top**, or **Bottom**. • centered in relation to the anchor control — click **Align** ▸ **Center**, and then click **Vertical** or **Horizontal**. • spaced in relation to each other — click **Align** ▸ **Space Evenly**, and then click **Vertical** or **Horizontal**. **TIP:** You can make the selected controls the same size as the anchor control by clicking **Align** ▸ **Make Same Size** and then clicking **Height**, **Width**, or **Both**.

You can also	
Use a grid to position controls	Display the grid by clicking **View** ▸ **Show Grid**, and click **View** ▸ **Snap to Grid** to force controls to align with the points on that grid. **TIP:** If you want to specify the amount of space between grid points on each axis, click **View** ▸ **Grid Options**.

To assign behaviors to one or more controls by using the Dialog Editor

- With the dialog box open for editing in the Dialog Editor, assign behaviors to one or more controls by using any of the methods in the following table.

To	Do the following
Specify the control that has focus when the dialog box opens	Click **Dialog** ▸ **Initial Focus**. In your dialog box, click the desired control. Click **OK** in the **Initial Focus** dialog box to apply your changes.
Define related controls as a group	Click **Dialog** ▸ **Control Groups**. In your dialog box, click the desired controls. Click **OK** in the **Control Groups** dialog box to apply your changes. **NOTE:** To create a functioning group of controls, you must set their order. You must also draw a group box around them (see "Using group boxes in dialog boxes" on page 115).
Specify the controls that you want the user to be able to tab through	Click **Dialog** ▸ **Tab Stops**. In your dialog box, click the desired controls. Click **OK** in the **Tab Stops** dialog box to apply your changes, and then click **Dialog** ▸ **Control Order**. In your dialog box, click the controls in the desired order. Click **OK** in the **Control Order** dialog box to apply your changes.
Specify the default button for a control, which is activated when the user presses **Enter** on that control	Click **Dialog** ▸ **Default Button**. In your dialog box, click the desired control, and then click the button that you want as the default for that control. Click **OK** in the **Default Button** dialog box to apply your changes.

Setting up callbacks for dialog boxes

As previously discussed (see "Understanding callbacks" on page 74), you can create callbacks for dialog boxes. Using a dialog-box callback lets the macro gather information from an active dialog box, rather than waiting until the dialog box is closed to gather that information. For example, you can use a dialog-box callback to return the value of a control without having to close the dialog box, or to refresh the contents of a control without having to destroy and reopen the dialog box.

Creating dialog-box callbacks

There are two requirements for creating a dialog-box callback.

The first requirement for creating a callback is to specify the name of the label to which to send callback messages. This label name is specified in the third parameter of the `DialogShow` command. If you need to watch for certain events, you must create statements for those events inside this label. At the end of the routine, a mandatory Return command ensures that the callback loop functions properly.

> The name in the label parameter of the `DialogShow` command may refer to either a label or a procedure. This parameter creates an implicit array of eleven elements, which has the same name as that label or procedure. The messages that are sent to the callback are accessed through these array elements.
>
> The actual `Label...Return` structure can be placed anywhere in your macro. It is better, however, to create a section in your macro just for labels, functions, and procedures.

The second requirement for creating a callback is to create a callback-message loop, which is necessary for trapping dialog-box events. The loop holds control of the macro until the loop is terminated. You can create a callback-message loop either by using the `CallBackWait` command or by using a `While` loop.

The following code segment shows a skeleton callback function:

```
DialogShow(1000; "WordPerfect"; Msgs)

CallbackWait

// This command initializes the loop.

Quit

Label(Msgs)

// This label identifies where dialog box events are watched.

... statements ...

Return
```

When a dialog box is displayed and active, messages from events in the dialog box are sent to the callback label. You can activate a callback loop in several ways, such as the following:

- pressing Alt + F4
- double-clicking the system-menu box
- clicking a button, radio-button, check-box, hot-spot, or scroll-bar control

Understanding callback loops

The following three examples illustrate different methods of creating callback loops.

These three examples watch for two events, or messages: the system closing, or the user clicking **OK**. (Both messages are handled identically in these examples, but in a real-world example, the actions could be quite different.) In these examples, either the dialog box is destroyed and the `CallBackResume` is sent, or the `Loop` variable is set to `False`. In either case, the callback loop ends, and the macro continues to the command that is directly after the `CallBackWait` command or the `While` loop. (In these examples, that next command is `Quit`, so the macro is ended.

The following example uses the `CallBackWait` and `CallBackResume` commands to create a callback loop. The `CallBackWait` command creates the loop for the callback. While this loop is active, the dialog box waits for an event to trigger the callback. Most controls activate the callback, or send messages through the loop to the callback label. Any messages are handled in the `Label` section of the code.

```
DialogShow ("Dialog1"; "WordPerfect"; Msg)
CallbackWait()
Quit

Label (Msg)
If (Msg[5] = 274)
DialogDestroy ("Dialog1")
CallbackResume()
Return
Endif
If (Msg[3] = "OKBttn")
DialogDestroy ("Dialog1")
```

```
CallbackResume()
Return
Endif
Return
```

The following example uses a `While` loop to create a callback loop. This method is not as efficient or easy as using the `CallBackWait` and `CallBackResume` commands.

```
DialogShow ("Dialog1"; "WordPerfect"; Msg)
Loop = TRUE
While (Loop)
EndWhile
Quit

Label (Msg)
If (Msg[5] = 274)
DialogDestroy ("Dialog1")
Loop = FALSE
Return
Endif
If (Msg[3] = "OKBttn")
DialogDestroy ("Dialog1")
Loop = FALSE
Return
Endif
Return
```

The following example uses a procedure to create a callback loop.

```
DialogShow("Dialog1"; "WordPerfect"; Msg; "OKBttn")
CallBackWait()
QUIT

Procedure Msg()
Switch (Msg[5])
CaseOf 274:
```

```
// "Close" was selected.
DialogDestroy("Dialog1")
CallBackResume
CaseOf 273:
// The macro cannot get to callback array element 3 unless
// array element 5 is 273.
Switch (Msg[3])
// The user chooses a control on the dialog box.
CaseOf "OKBttn":
// "OK" was watched for.
DialogDestroy("Dialog1")
CallBackResume()
// ...Add additional CaseOf statements for additional controls.
EndSwitch
EndSwitch
Return
EndProcedure
```

Using region commands to specify and return dialog-box values

A "region" is the name of a dialog-box control. A region name is generally made up of the name of the dialog box and of the control in question. Region names are case-sensitive.

Consider the following lines of code:

```
DialogDefine ( "MainDialog"; 50; 50; 200; 50; OK! | Cancel! _
| Percent!; "My Main Dialog" )
DialogAddText ( "MainDialog"; "TextControl1"; 10; 10; 180; 9; _
Left!; "Make a selection:" )
DialogShow ( "MainDialog"; "WordPerfect" )
```

In the preceding code, the dialog box called `MainDialog` has been defined with only one control — a text control. Both the dialog box and the text control have a region name. The region name for the dialog box is `"MainDialog"`, and the region name for the control is `"MainDialog.TextControl1"`.

Note that the region name of the control (`MainDialog.TextControl1`) contains a period (`.`). The period is used to narrow the region. Whenever you refer to a region (that is, a dialog-box name and control name) in a dialog box, the period is used. The only time the period is not used is when it is only the dialog box that is being referred to, as in the following example:

```
RegionSetWindowText ( "MainDialog"; "A New Dialog Title" )
```

In the preceding example, the title text of the dialog box is changed to `"A New Dialog Title"`. We do not need a narrowed region.

The need for narrowed regions becomes apparent when a macro contains multiple dialog boxes. Assume that a macro had two dialog boxes, `Dialog1` and `Dialog2`, and that each dialog box had a text control called `"Text1"`. If we needed to change the text for that control, we would use the `RegionSetWindowText` command. Consider the following region command:

```
RegionSetWindowText ( "Text1"; "New Control Text" )
```

In the preceding region command, it is unclear which of the two controls is being referred to. (In fact, as written, the preceding code would generate a run-time error because the dialog box is not specified.) By prefacing the control name by the dialog box name, we can narrow the region to specify which dialog box and which control needs to be changed, as in the following corrected code:

```
RegionSetWindowText ( "Dialog1.Text1"; "New Control Text" )
```

The preceding region command would not generate an error. In this case, the text of control `"Text1"` would be changed in `"Dialog1"`.

When naming dialog boxes and dialog box controls, use names that are descriptive. Descriptive names make it easier to remember what a control is used for, and they make your macro code more understandable. This usefulness of this guideline becomes especially clear if you want to expand your macro in the future.

In a callback dialog box, region commands can be used to update, change, query, and set values while that dialog box is active. The primary location to use region commands is in the callback loop. For example, if a dialog box contains a list box, and you want to trap the double-click event, you could write the following section of code:

```
DialogDefine ( "MainDialog"; 50; 50; 100; 125; OK! | Cancel! |_
Percent!; "Selection Dialog" )
DialogAddListBox ( "MainDialog"; "ListBox1"; 10; 10; 80; 95; _
Sorted!; lbVar )
DialogAddListItem ( "MainDialog"; "ListBox1"; "Pear")
DialogAddListItem ( "MainDialog"; "ListBox1"; "Apple")
DialogAddListItem ( "MainDialog"; "ListBox1"; "Banana")
DialogAddListItem ( "MainDialog"; "ListBox1"; "Orange")
DialogShow ( "MainDialog"; "WordPerfect"; Msg )
CallBackWait ()
DialogDestroy ( "MainDialog" )
Quit

Label ( Msg )
If ( Msg[5] = 274 OR Msg[3] = "CancelBttn" )
Quit()
EndIf

If ( Msg[3] = "OKBttn" )
vSelectedItem := RegionGetSelectedText( "MainDialog.ListBox1" )
Type ( vSelectedItem )
CallBackResume()
EndIf

If ( Msg[3] = "ListBox1" AND Msg[10] = 2 )
vSelectedItem := RegionGetSelectedText( "MainDialog.ListBox1" )
MessageBox ( vRetVar; "Selected Item"; vSelectedItem + " - was _
selected!")
EndIf
Return
```

As the preceding example demonstrates, you can use region commands to design sophisticated macros. A programmer who understands the task that needs to be performed can determine how many region commands are necessary.

Testing dialog boxes

Before you start putting your dialog box to use within its macro, it's a good idea to test it. Testing a dialog box helps you ensure that it functions correctly.

If you created your dialog box by using PerfectScript code, you must test it by using the PerfectScript Debugger. For information, see "Debugging macros" on page 135.

If you created your dialog box by using the PerfectScript Dialog Editor, you can test it from directly within the Dialog Editor. For information, see "To test a dialog box by using the Dialog Editor" on page 131.

To test a dialog box by using the Dialog Editor

1 With the dialog box open for editing in the Dialog Editor, click **Dialog ▸ Test**.
2 Use each control on the dialog box, and make note of any necessary changes.
3 Close the dialog box to return to the Dialog Editor.
4 Make the necessary changes to the dialog box.
5 Repeat steps 1 to 4 until the dialog box functions as desired, and then save the dialog box.

Displaying dialog boxes

Regardless of whether you create a dialog box by manually coding it or by using the Dialog Editor, you must manually code the procedure for displaying that dialog box in your macro. The process of displaying a dialog box involves four steps:

- opening the dialog box
- releasing the dialog box
- closing the dialog box
- destroying the dialog box

For step-by-step information on displaying a dialog box in a macro, see "To display a dialog box in a macro" on page 134.

Opening dialog boxes by using PerfectScript code

You can use the `DialogShow` command to open a dialog box in a macro.

Before a dialog box can be manipulated by using a `Region` command, that dialog box must be loaded into memory. The `DialogShow` command both loads the dialog box into memory and opens it. (By contrast, the `DialogLoad` command loads the dialog box into memory but does not open it.)

The following is an example of the `DialogShow` command:

```
DialogShow("DialogName";"WordPerfect";CallBack@)
```

The first parameter, `Dialog`, is required. This parameter specifies the name of the dialog box to be opened. (In the preceding example, this name is `DialogName`).

The second parameter, `Parent`, is optional. This parameter specifies the named region of the parent window for the macro dialog box. (In the preceding example, the `WordPerfect` window is the parent window for the dialog box.) Named regions are defined by the application. The region consists of the application name, followed by a period (`.`), followed by additional words that narrow the named region to the appropriate window. For example, the named region of the document window in WordPerfect is `WordPerfect.Document`.

Names for dialog boxes and controls are case-sensitive, so be sure to correctly reference them in your macros.

The third parameter, `Callback`, is optional. This parameter specifies a label that identifies a callback function. If you do not specify a callback parameter in the `DialogShow` command, the macro does not execute until you dismiss the dialog box. If you use a callback, the macro executes while the dialog box is displayed. It is up to the callback to prevent the macro from terminating prematurely and to shut down the macro dialog box by using the `DialogDismiss` command.

A fourth parameter not shown in the preceding example, `Focus`, is optional. This parameter can be used to specify the control that receives initial focus when the dialog box is opened.

For more information about the `DialogShow` command, please see the DialogShow topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Releasing dialog boxes by using PerfectScript code

You can release a dialog box by performing one of several actions. The action that you perform is returned, as a value, to an implicit variable `MacroDialogResult`.

Clicking the **OK** button returns a value of `1`, which instructs the macro to apply the changes that you made to the dialog box.

Performing one of the following actions returns a value of `2`, which instructs the macro to ignore the changes that you made to the dialog box.

- clicking the **Cancel** button
- clicking the **Close** button on the system-menu box
- double-clicking the system-menu box
- pressing **Alt + F4**

Clicking a user-defined button or a user-defined hot spot returns the value of the **Control** parameter of that user-defined item.

For more information about the `MacroDialogResult` variable, please see the MacroDialogResult topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Closing dialog boxes by using PerfectScript code

You can use the `DialogDismiss` command to close a dialog box in a macro (and clear the value of the implicit variable `MacroDialogResult`.)

The following is an example of the `DialogDismiss` command:

```
DialogDismiss("DialogName";"OKBttn")
```

The first parameter, `Dialog`, is required. This parameter specifies the name of the dialog box to be closed.

The second parameter, `Control`, is required. This parameter specifies the named region of the control that is used to close the dialog box.

Names for dialog boxes and controls are case-sensitive, so be sure to correctly reference them in your macros.

If your macro needs to refer to the value of the `MacroDialogResult` variable for a dialog box, assign that value to another variable before executing the `DialogDismiss` command.

If you need to reopen a dialog box, use the `DialogShow` command.

For more information about the `DialogDismiss` command, please see the DialogDismiss topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Destroying dialog boxes by using PerfectScript code

You can use the `DialogDestroy` command to destroy a dialog box from memory. Destroying any unused dialog boxes in your macro is a good way to free up memory.

You cannot destroy dialog boxes that were created by using the Dialog Editor.

The `DialogDestroy` command has one parameter, `Dialog`, which is required. This parameter specifies the name of the dialog box to be destroyed.

For more information about the `DialogDestroy` command, please see the DialogDestroy topic in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

To display a dialog box in a macro

1 Open the macro for editing.

2 Type or insert the `DialogShow` command where you want the dialog box to open in the macro, and specify the following parameters for that command:
- `Dialog` — to specify the name of the dialog box
- `Parent` (optional) — to specify the named region for the parent window of the dialog box
- `Callback` (optional) — to specify a callback parameter, if you want the macro to execute while the dialog box is open
- `Focus` (optional) — to specify the name of the dialog-box control to receive initial focus

REMEMBER: Dialog-box names are case-sensitive.

3 Assign the value of the implicit variable `MacroDialogResult` to some other variable if you want to capture the value that it receives when the dialog box is released.

REMEMBER: If you close a dialog box by using a **Cancel** button, a control other than a push-button, or a non-existent control, your changes to the dialog box do not take effect. However, if you use a push-button other than a **Cancel** button, the variable values are set and your changes take effect when you dismiss the dialog box.

4 Type or insert the `DialogDismiss` command after a `DialogShow` command that uses callback, and specify the following parameters for that command:
- `Dialog` — to specify the name of the dialog box
- `Control` — to specify the named region of the control that is used to close the dialog box

REMEMBER: Dialog-box names and control names are case-sensitive.

5 Type or insert the `DialogDestroy` command if you want to free up memory by destroying the dialog box.

REMEMBER: You cannot destroy dialog boxes that were created by using the Dialog Editor.

Debugging macros

To ensure that your macros work as expected, it's important to debug them by using the PerfectScript Debugger.

This section contains the following topics:

- Getting started with the PerfectScript Debugger
- Using the Debugger to debug macros
- Getting more information while debugging macros
- Working with breakpoints while debugging macros
- Working with variables while debugging macros
- Navigating the code while debugging macros
- Troubleshooting the Debugger

Getting started with the PerfectScript Debugger

When a macro is played or compiled, it is examined by the PerfectScript Debugger. The Debugger is designed to help you find and correct errors and other problems in your macros.

The Debugger workspace has the following features:

- a menu — provides access to all the commands for debugging macros. For more information, see "Using the Debugger menu" on page 136.
- a toolbar — provides access to common features for debugging macros. For more information, see "Using the Debugger toolbar" on page 136.
- the **State** line — indicates whether the Debugger is active and, if so, why. For more information, see "Using the State line in the Debugger" on page 136.
- the **Source** list — displays the source of the macro being debugged. For more information, see "Using the Source list in the Debugger" on page 137.
- the **Call History** list — displays, in reverse order, the user-defined subroutines that are called by the macro. For more information, see "Using the Call History list in the Debugger" on page 137.

- the **Variables** list — displays the variables that are accessible to the macro at the location indicated in the **Call History** list. For more information, see “Using the Variables list in the Debugger” on page 138.

The Debugger also provides several information windows, which provide details about the macros that you debug. For information, see “Getting more information while debugging macros” on page 143.

For information about any control in the Debugger, click ? (or press **Shift + F1**), and then click the control.

Using the Debugger menu

The Debugger menu gives you access to all the commands for debugging macros.

The Debugger menu is always displayed.

Using the Debugger toolbar

The Debugger toolbar gives you instant access to a wide range of features for debugging macros.

You can display or hide the Debugger toolbar by clicking **View ▸ Toolbar**. A check mark next to the **Toolbar** command indicates that the toolbar is displayed.

You can customize the Debugger toolbar by right-clicking any blank area on the toolbar. The **Customize Toolbar** dialog box is displayed, from which you can remove, add, or reorder the toolbar buttons. The toolbar configuration that you choose is applied when you debug all other macros.

Using the State line in the Debugger

The **State** line indicates whether the Debugger is active and, if so, why.

While the Debugger is active (for example, when a breakpoint is on the start of a macro statement, or when an error is incurred) the execution of the macro is suspended. You are therefore prevented from interacting with any displayed prompts, message boxes, or dialog boxes.

When a macro is playing, the **State** line reads, “Macro is running,” and the Debugger is inactive. Although you can use some Debugger features (for example, to set breakpoints) while the Debugger is inactive, you cannot use any features that access information about the state of the macro.

The **State** line is always displayed.

Using the Source list in the Debugger

The **Source** list displays the source of the macro being debugged (as taken from the listing file for the compiler).

You can display the source of another macro file (such as a **Use** file) by clicking **File ▸ Open**. The last nine accessed macro files are listed in the **File** menu.

The left margin of the **Source** list displays the following items:

- a red arrow — indicates the next line that the macro will execute
- an indicator — shows which statements have breakpoints, and whether those breakpoints are enabled or disabled

You can double-click any line that contains a macro statement to place a breakpoint on that line.

You can also enable or disable any defined breakpoint.

When you point to a variable, label, token, or command in the **Source** list, a floating tip displays details about that item. If the Debugger cannot identify the item — such as a variable that has not yet been defined, or a label defined in a **Use** file that has not yet been loaded — two question marks (??) are displayed.

The **Source** list is always displayed.

You can right-click the **Source** list to display a context-sensitive menu of relevant commands.

You can adjust the size of the **Source** list by dragging its split bar. Alternatively, you can select the split bar and use the arrow key to move it by 1 pixel; hold the **Shift** key to move the selected split bar by 5 pixels, or hold the **Ctrl** key to move it by 10 pixels.

Using the Call History list in the Debugger

The **Call History** list displays the user-defined subroutines (labels, functions, and procedures) that are called by the macro. The name of each subroutine is displayed, along with the line number where execution within that subroutine was interrupted, and the file in which the subroutine is contained. The subroutines are listed in reverse order, so the current location is provided at the top of the list.

When you select an entry in the **Call History** list, the source for that macro is displayed in the **Source** list, and the associated line is highlighted and indicated by a green triangle in the left margin (unless the top entry is selected, in which case the red arrow is displayed). The variables accessible to the macro at that point are then displayed in the **Variables** list below.

For more information about subroutines, see “Understanding subroutines” on page 59.

The **Call History** list is always displayed.

You can right-click the **Call History** list to display a context-sensitive menu of relevant commands.

You can adjust the size of the **Call History** list by dragging its split bar. Alternatively, you can select the split bar and use the arrow key to move it by 1 pixel; hold the **Shift** key to move the selected split bar by 5 pixels, or hold the **Ctrl** key to move it by 10 pixels.

Using the Variables list in the Debugger

The **Variables** list displays, by name, the variables that are accessible to the macro at the location indicated in the **Call History** list. Also displayed are the following:

- the pool type for each variable (**Local**, **Global** or **Persistent**)
- the type of value that each variable contains
- the current value of each variable

Arrays are a form of variables, and so they, too, are displayed in the **Variables** list — along with their declared dimensions and a **Contents** type of array. You can expand array variables to display the individual array elements in the list, or you can collapse them to hide the individual elements; in this way, you can examine the individual elements of the array as normal variables.

If a variable is an address (alias) parameter to a user-defined subroutine, its **Contents** type is displayed as **Alias**, and it may be expanded and collapsed like an array to show the actual variable to which it is mapped.

If an alias variable is mapped to a **Global** or **Persistent** variable, then the variable-pool type is displayed appropriately. However, if it is a **Local** variable, then the pool type is displayed as **Local to Caller**, to distinguish it from a variable that is local to the current subroutine.

The current sort column and sort order are indicated by a greater than symbol (>) or a less than symbol (<) before the name of the column.

If the **Variables** list is sorted by variable name or by pool, expanded array elements are kept with their corresponding array. Sorting by the other columns may separate array elements from each other, depending on the contents of the array element.

For more information about variables, see “Understanding variables” on page 10. For more information about working with variables in the Debugger, see “Working with variables while debugging macros” on page 156.

The **Variables** list is always displayed.

You can right-click the **Variables** list to display a context-sensitive menu of relevant commands.

You can adjust the size of the **Variables** list by dragging its split bar. Alternatively, you can select the split bar and use the arrow key to move it by 1 pixel; hold the **Shift** key to move the selected split bar by 5 pixels, or hold the **Ctrl** key to move it by 10 pixels.

Using the Debugger to debug macros

You can use the PerfectScript Debugger to debug macros from within the PerfectScript utility.

You can also debug macros from directly within WordPerfect, Quattro Pro, or Presentations. For information, please see the Help file for the application.

Setting up the Debugger

Before you begin debugging macros, it’s a good idea to set up the Debugger.

You can decide whether to display *source code* for each macro that you debug.

You can remove the source from a macro, even if that macro was compiled from an earlier version of PerfectScript. The act of removing the source from a macro cannot be reversed.

You can also protect a macro from being accidentally edited in WordPerfect. When a macro is protected, it can be compiled and played, but not opened, in WordPerfect. (Any attempt to open a protected macro in WordPerfect generates an “unknown file type” error.) The act of protecting a macro can be reversed.

For more information about removing the source from a macro or protecting a macro, please see the WordPerfect Help.

You can also decide whether to generate a *listing file* for each macro that you debug. The listing file makes a copy of each procedure in the macro and numbers each line of code, allowing you to tell which part of the macro corresponds to each line number. Any error messages and warnings that arose during compilation are provided at the bottom of the listing file. By using a listing file, you can more easily find and correct the errors caught by the Debugger.

The listing file has the same name as the original macro, but with a **.wcl** extension.

If you want to see which macro lines correspond to which line numbers, you can display the listing file in any ASCII-based text editor.

You can specify the settings for invoking the Debugger.

You can also specify whether create *event logs* while debugging macros.

For more information, see “Logging events for breakpoints” on page 152.

Finally, you can specify settings for *animating* macros. Animation lets you step through macros line-by-line, automatically invoking the Debugger at each step so that you can check the variables and other calls.

For more information, see “Animating macros” on page 160.

For more information about setting up the Debugger, see “To set up the Debugger” on page 141.

Debugging macros

Before you can debug a macro, you must compile it — and any other macro files that it “uses” or “runs” or “chains” to. Compiling a macro for debugging adds necessary information to the macro. For information about using the PerfectScript utility to compile macros, see “Compiling macros” on page 96.

A compiled macro can be debugged by playing it in the Debugger. For information about this process, see “To debug a macro by using the Debugger” on page 142.

The Debugger uses configuration (DBG) files to store macro-related information.

The common configuration file for the Debugger is called **startup.dbg**. This file stores information about the Debugger, such as the following details:

- which macro filenames are stored in the "most recently used" list
- which information windows are open (see "To display or hide an information window" on page 148)
- which variable pools are displayed (see "To display variables while debugging a macro" on page 157)

When you debug a macro, the Debugger loads the settings that are stored in the **startup.dbg** file, and it checks the version number of the PerfectScript system against the version number of the macro system. All of this information is used to create a macro-specific configuration file, which is given a **.dbg** extension and stored in the same folder as the macro.

To set up the Debugger

1 In the PerfectScript utility, click **Tools ▸ Settings**.

2 Click the **Compile** tab, and enable any of the following settings:
- **Include debug information** — displays source code for each macro that you debug
- **Generate listing file** — generates a listing file for each macro that you debug

3 Click the **Debug** tab, and enable any of the following settings:
- **Invoke debugger on macro start** — opens the Debugger when a macro is started
- **Invoke debugger on errors** — opens the Debugger when a macro error is encountered
- **Enable debugger event logging** — allows the Debugger to log events while debugging macros

4 In the **Animate** area of the **Debug** page, do the following:
- Enable the **'RunTo' does 'Step Into'** option if you want to step through macros one statement at a time, or enable the **'RunTo' does 'Step Over'** option if you want to step through macros one subroutine at a time. For more information about these options, see "Stepping through macros" on page 160.
- In the **Delay (seconds)** box, specify the number of seconds for the macro to pause after executing each step.

To debug a macro by using the Debugger

1 In the PerfectScript utility, click **File ▸ Debug ▸ Play**.

2 Select the macro that you want to debug, and click **Debug**.

If you are prompted to specify the listing file for the macro, do one of the following:

- Specify the listing file. Select the listing file, and then click **Open**.
- Decline to specify the listing file. Click **Close**.

3 Do any of the following:

- Display any desired *information windows*. For information, see "Getting more information while debugging macros" on page 143.
- Specify the *breakpoint*-related settings for the macro. For information, see "Working with breakpoints while debugging macros" on page 150.
- Specify the *variable*-related settings for the macro. For information, see "Working with variables while debugging macros" on page 156.

4 Click **Debug ▸ Continue** to play the macro through to the next stopping point, or choose another option for navigating the macro code. For information, see "Navigating the code while debugging macros" on page 159.

5 Use the specified macro editor to correct any errors.

For information about debugging macros that stop at error messages or that contain faulty callbacks, see "Troubleshooting the Debugger" on page 162.

You can also

Compile a macro by using the Debugger	In the PerfectScript utility, click **File ▸ Debug ▸ Compile**. Select the macro that you want to compile, and click **Compile**.
Pause the debugging of a macro	Do one of the following: • Click **Debug ▸ Break**. • Press **Ctrl + F5**. **NOTE:** This feature interrupts the macro and activates the Debugger, giving you access to various features in the Debugger.

You can also

Restart the debugging of a macro	Do one of the following: • Click **Debug** ▸ **Restart**. • Press **Shift + F5**. **NOTE:** When a macro is restarted, all variables created by the macro — except persistent variables — are deleted. In addition, the Debugger begins at the top of the macro and resets all state information about the macro to its original conditions.
Stop the debugging of a macro	Do one of the following: • Click **Debug** ▸ **Stop Debugging**. • Press **Alt + F5**. **NOTE:** When a macro is stopped, the Debugger closes. All defined breakpoints, watched variables, and opened macro files are stored in a Debugger configuration file for that macro. This file is given a **.dbg** extension, stored in the same folder as the macro, and loaded the next time you debug that macro.

Getting more information while debugging macros

The Debugger provides several windows that you can use to display various types of information about the current state of a macro. These windows are refreshed whenever the Debugger becomes active. Most of the windows display information that is specific to the current execution point in the macro, but by selecting a different entry in the **Call History** list of the Debugger (see “Using the Call History list in the Debugger” on page 137), you can display information that is specific to the selected entry.

Each information window has a context menu that lets you navigate among the other information windows, the main Debugger window, and any matching macro source line. By double-clicking an item in an information window, you can jump to the position of that item in the macro (and highlight that item with a gray arrow in the left margin of the **Source** list).

The Debugger provides the following information windows:

- **Label Table** window — lists all labels that are defined at the execution point that is selected in the **Call History** list. For more information, see "Using the Label Table window in the Debugger" on page 144.
- **Use File Table** window — lists all **Use** files that are referenced in `Use` statements by the macro file that is selected in the **Call History** list. For more information, see "Using the Use File Table window in the Debugger" on page 145.
- **Product Table** window — lists all applications and products that have commands in the macro file that is selected in the **Call History** list. For more information, see "Using the Product Table window in the Debugger" on page 145.
- **Dialog List** window — lists all user-created macro dialog boxes that are currently defined or that exist in the prefix packet of the macro file that is selected in the **Call History** list. For more information, see "Using the Dialog List window in the Debugger" on page 145.
- **Condition Handlers** window — lists all condition handlers that are defined for the execution point that is selected in the **Call History** list. For more information, see "Using the Condition Handlers window in the Debugger" on page 146.
- **Macro Info List** window — lists all data that can be obtained from the `MacroInfo` command for a macro when an execution point is selected in the **Call History** list. For more information, see "Using the Macro Info List window in the Debugger" on page 147.
- **Callback Queue** window — lists all items in the callback queue and indicates which callbacks are currently active and which are pending. For more information, see "Using the Callback Queue window in the Debugger" on page 147.
- **Macro Header** window — displays the object-header information for the macro file. For more information, see "Using the Macro Header window in the Debugger" on page 147.

For information about using the information windows, see "To display or hide an information window" on page 148.

Using the Label Table window in the Debugger

The **Label Table** window lists all labels that are defined at the execution point that is selected in the **Call History** list.

For each label, the following items are displayed:

- the name of the label
- the type of the label — which is either **Local** or **Global**. **Local** labels are defined by the `Label` statement in a macro, and they are visible only within the function or

procedure where they are defined. **Global** labels are user-defined functions and procedures, and they are visible anywhere in a macro file, as well as in other macro files that have a `Use` statement of the file containing the function or procedure.

- the line number of the source line where the label (or function or procedure) is defined
- the name of the file where the label (or function or procedure) is defined

The **Label Table** window is a modeless dialog box.

Using the Use File Table window in the Debugger

The **Use File Table** window lists all **Use** files that are referenced in `Use` statements by the macro file that is selected in the **Call History** list.

If the labels for that **Use** file have been loaded (as happens the first time a function or procedure is called from the **Use** file), then the **Loaded** column shows **True**.

The **Use File Table** window is a modeless dialog box.

Using the Product Table window in the Debugger

The **Product Table** window lists all applications and products that have commands in the macro file that is selected in the **Call History** list.

If an `Application` statement for an application is in a macro, but the macro does not actually contain any commands for that application, that application is not displayed in this list.

Each application or product that is listed displays the version number of the PID (product interface description) file that was used when this macro was compiled. This version number is used to determine if a compiled macro has become out-of-date when a new version of an application is installed and the macro is played.

The **Product Table** window is a modeless dialog box.

Using the Dialog List window in the Debugger

The **Dialog List** window lists all user-created macro dialog boxes that are currently defined or that exist in the prefix packet of the macro file that is selected in the **Call History** list.

For each dialog box, the following items are displayed:

- name
- state

- type — which is either **Text** or **Binary**. **Text** dialog boxes are defined by using `DialogDefine` and `DialogAdd` statements in a macro, while **Binary** dialog boxes are created by using the Dialog Editor (see "Understanding dialog boxes" on page 101) and are stored in the prefix packet area of a macro file.
- callback label — which is displayed if the dialog box is currently showing and a callback label was specified
- position and size (at creation, not current)
- style — which is defined in the `DialogDefine` statement or in the Dialog Editor

The states that are available to a dialog box depend on its state, as follows:

- **Defined** (**Text** dialog boxes only) — means that the dialog box has been defined by a `DialogDefine` statement but hasn't been loaded or shown yet
- **In Prefix** (**Binary** dialog boxes only) — means that the dialog box was found in the prefix packet of the current macro file but hasn't been loaded or shown yet
- **Loaded** (**Text** and **Binary** dialog boxes) — means that the dialog box has been loaded by a `DialogLoad` statement or by a `Region` command
- **Showing** (**Text** and **Binary** dialog boxes) — means that the dialog box is currently showing by a `DialogShow` statement

In the lower half of the **Dialog List** window, the list of controls defined for the selected dialog box are displayed. For each control, the following details are given:

- order
- name
- type
- position and size (at creation, not current)
- associated variable
- associated style
- associated data

The **Dialog List** window is a modeless dialog box.

Using the Condition Handlers window in the Debugger

The **Condition Handlers** window lists all condition handlers that are defined for the execution point that is selected in the **Call History** list.

For each condition handler, its action and data are displayed. The standard condition handlers — such as `Error`, `Cancel`, and `NotFound` — are displayed in this list, as well as handlers for callbacks such as `OnDDEAdvise` and callback dialog boxes.

The **Action** column provides information about whether the condition causes the macro to abort or quit, or whether the condition causes a label to be called or jumped to. If the handler has been disabled, the **Ignore** action is displayed, indicating that the abort, call, or jump will be ignored.

The **Condition Handlers** window is a modeless dialog box.

Using the Macro Info List window in the Debugger

The **Macro Info List** window lists all data that can be obtained from the `MacroInfo` command for a macro when an execution point is selected in the **Call History** list. This data can include labels, line numbers, and filenames.

The **Macro Info List** window is a modeless dialog box.

See also the Help for the MacroInfo command in the PerfectScript Command Reference section of the PerfectScript Help file (**psh.chm**).

Using the Callback Queue window in the Debugger

The **Callback Queue** window lists all items in the callback queue and indicates which callbacks are currently active and which are pending.

It is possible for multiple callbacks to be active at the same time.

The callback queue contains entries for callback dialog boxes and for `OnDDEAdvise` notifications. The label to be called by each callback is specified.

The **Status** column indicates whether this callback is for notification only, or whether the callback can affect the action that is performed by the macro system when the callback is complete. Callbacks are always for notification only.

The contents of the data array for the callback are also displayed, as is (where possible) an interpretation of the specific array elements.

The **Callback Queue** window is a modeless dialog box.

Using the Macro Header window in the Debugger

The **Macro Header** window displays the object-header information for the macro file, including the version number of the macro system that was used to compile this macro file.

The **Macro Header** window is a modal dialog box.

To display or hide an information window

- While debugging a macro, do any of the following:

To display or hide the following	Do the following
Label Table window	Click **View** ▸ **Label Table** (or press **Alt** + 1). A checkmark next to the **Label Table** command indicates that all labels that are defined at the execution point are displayed. **TIP:** You can double-click a label to display the macro file in the **Source** list and highlight the line containing the definition for that label.
Use File Table window	Click **View** ▸ **Use File Table** (or press **Alt** + 2). A checkmark next to the **Use File Table** command indicates that all **Use** files that are referenced in `Use` statements by the macro file are displayed. **TIP:** You can load a **Use** file into the **Source** list by double-clicking it.
Product Table window	Click **View** ▸ **Product Table** (or press **Alt** + 3). A checkmark next to the **Product Table** command indicates that all applications and products that have commands in the macro file are displayed.
Dialog List window	Click **View** ▸ **Dialog List** (or press **Alt** + 4). A checkmark next to the **Dialog List** command indicates that all user-created macro dialog boxes are displayed. **TIP:** You can double-click a dialog box to display, in the **Source** list, the macro file where the callback label is defined, and to highlight the source line that contains the label definition.

To display or hide the following	Do the following
Condition Handlers window	Click **View** ▶ **Condition Handlers** (or press **Alt** + 5). A checkmark next to the **Condition Handlers** command indicates that all defined condition handlers are displayed. **TIP:** If a label is associated with a condition handler, you can double-click the item to display, in the **Source** list, the macro file where that label is defined and to highlight the source line that contains the label definition.
Macro Info List window	Click **View** ▶ **Macro Info List** (or press **Alt** + 6). A checkmark next to the **Macro Info List** command indicates that all data that can be obtained from the `MacroInfo` command is displayed. **TIP:** You can double-click an item to display, in the **Source** list, the macro file, and to highlight the source line that contains the label definition or line number.
Callback Queue window	Click **View** ▶ **Callback Queue** (or press **Alt** + 7). A checkmark next to the **Callback Queue** command indicates that all pending items in the callback queue are displayed. **NOTE:** You can remove a selected callback from the queue window by pressing **Delete** or **Backspace**. However, a warning prompts you to confirm the action because deleting a callback can dramatically alter the behavior of the macro. **TIP:** You can double-click a line to display, in the **Source** list, the macro file, and to highlight the source line that contains the label definition.

To display or hide the following	Do the following
Macro Header window	Click **View** ▶ **Macro Header** (or press **Alt + 8**). A checkmark next to the **Macro Header** command indicates that all object-header information for the macro file is displayed. **NOTE:** The **Macro Header** window displays data regardless of whether the macro is protected or whether it contains source code.

You can also	
Jump to the associated source line from within an information window	Press **Space.**
Activate the main Debugger window	Press **Ctrl + Home.**
Activate the next information window	Do one of the following: • Click **View** ▶ **First/Next window**. • Press **Ctrl + F6**. • Press **Ctrl + Down Arrow**.
Activate the previous information window	Do one of the following: • Click **View** ▶ **Last/Previous window**. • Press **Ctrl + Shift + F6**. • Press **Ctrl + Up Arrow**.
Hide all information windows	Click **View** ▶ **Close All**.

Working with breakpoints while debugging macros

You can use breakpoints when debugging macros. When the Debugger encounters a breakpoint in a macro, the execution of that macro is interrupted and suspended, and the Debugger becomes active so that you can examine the state of the macro at that breakpoint.

You can manage breakpoints by using the **Breakpoints** dialog box.

Setting breakpoints

You can set breakpoints at any line, any DLL call, or any other place in a macro. The Debugger becomes active when it encounters a breakpoint in the macro, during which time you can check the macro code for labels, functions, procedures, variables, and so on.

The Debugger automatically creates three breakpoints: **Macro Start**, **Macro End**, and **Error**. These breakpoints, which are indicated by an exclamation mark (!), can be removed if desired.

The **Breakpoints** dialog box has a context menu that lets you sort breakpoints by the following columns: **Type**, **Location**, **Macro**, and **Pass Count**.

The **Type** column indicates the breakpoint type:

- **DLL Call** — breaks when the macro calls a DLL file
- **Error** — breaks when an error occurs while running the macro
- **Label/Routine Call** — breaks when a label or user-defined routine comes up in the macro. Everything that is not specifically contained in a routine or label is in the `<main>` routine.
- **Label/Routine Return** — breaks when a label or user-defined routine comes up in the macro, but stops before executing the return from the label call
- **Line Number** — breaks when the macro reaches a specified line number. This is the most common type of breakpoint: It suspends the execution of the macro and activates the Debugger when the specified line number is reached.
- **Product Call** — breaks when the macro makes a call to an application or product
- **Variable Access** — breaks when the macro accesses a variable
- **Variable Assign** — breaks when the macro assigns a value to a variable

The **Location** column indicates the location of the breakpoint in the macro code.

The **Macro** column indicates the macro to which the breakpoint applies. By default, a breakpoint applies to all the macro files that are used by the macro that you are debugging; however, you can limit the breakpoint to a single macro file.

The **Pass Count** column indicates the passcount for the breakpoint, which represents the number of times that the breakpoint conditions can occur before the breakpoint actually takes effect. The passcount decrements each time that the conditions occur, and the breakpoint triggers when the passcount reaches zero.

For more information about setting breakpoints, see “To set a breakpoint in a macro” on page 153.

Moving between breakpoints

You can move between the breakpoints in a macro.

For more information about moving between breakpoints, "To move between the breakpoints in a macro" on page 154.

Disabling breakpoints

You can temporarily disable all breakpoints. In this scenario, no breakpoints are recognized — even if their conditions occur — until all breakpoints are re-enabled, or until the macro ends and the Debugger terminates.

For more information about disabling breakpoints, see "To disable all breakpoints in a macro" on page 154.

Logging events for breakpoints

You can log events while debugging macros. The event log records an entry for each event that occurs during debugging; these entries include standard messages (such as "Debugger event logging enabled") and custom messages for which you supply the comment.

You can display the event log for a macro by clicking **View ▸ Event Log** or pressing **Alt + 9**.

All breakpoints allow a message to be logged to the event log (if enabled) when that breakpoint is triggered.

A breakpoint can be set to log a message, to cause a break in the macro, or both. In the left margin of the **Breakpoints** list, a hand symbol is displayed. A yellow hand indicates that a breakpoint that causes a break in the macro (and may also log an event message), while a blue hand indicates a breakpoint that logs an event message and does not cause a break in the macro.

For more information about creating event logs, see "To log events for the breakpoints in a macro" on page 155.

Executing tokens at breakpoints

When a macro is stopped at a breakpoint, you can execute any PerfectScript command (or "*token*") in a very localized temporary environment. The Debugger displays the commands along with their parameters and types, and it lets you select a command and specify a value for each parameter. When you execute the selected command, its return value is displayed (and assigned to a variable name, if one is specified).

Be careful when executing tokens. Some PerfectScript commands cause the internal state of the running macro to change and can therefore cause errors to occur later in the macro. (Most of these commands do not appear in the command list and cannot be selected.)

Values cannot be assigned to variables that have the same name as command token names.

PerfectScript supports handler DLLs for third-party tokens. All tokens (not just the PerfectScript ones) are passed to the `ValidateToken` entry point. If the third-party DLL does not accept or approve the token, it can cause PerfectScript to abort the macro. PerfectScript tokens are passed to the `HandleToken` entry point in the third-party DLLs.

For more information about executing tokens, see “To execute a token at a breakpoint in a macro” on page 155.

To set a breakpoint in a macro

1 While debugging a macro, do any of the following:
- Click **Debug** ▶ **Breakpoints** ▶ **Edit**.
- Right-click any line in the macro, and then click **Edit Breakpoints**.
- Press **Ctrl + B**.

2 Choose the type of breakpoint that you want to use from the **Type** list box, and then click **Add**.

3 Specify any settings for the breakpoint, and then click **Update**.

You can also

Set a line-number breakpoint in a macro	Do one of the following: • Double-click the line in the macro. • Select the line in the macro, and click **Debug** ▶ **Breakpoints** ▶ **Add**. • Select the line in the macro, and press **Insert**. • Right-click the macro, and click **Add Breakpoint**.

You can also

Remove a breakpoint from a macro	Do one of the following: • Double-click the breakpoint in the macro. • Select the breakpoint in the macro, and click **Debug** ▸ **Breakpoints** ▸ **Remove**. • Select the breakpoint in the macro, and press **Delete**. • Right-click the breakpoint in the macro, and click **Remove Breakpoint**.

To move between the breakpoints in a macro

- While debugging a macro, do one of the following:
  - Click **Edit** ▸ **Find Next Breakpoint** (or press **Ctrl + N**) to go to the next breakpoint.
  - Click **Edit** ▸ **Find Previous Breakpoint** (or press **Ctrl + P**) to go to the previous breakpoint.

To disable all breakpoints in a macro

- While debugging a macro, click **Debug** ▸ **Breakpoints** ▸ **Enable/Disable All Breakpoints**.

You can also

Disable a single line-number breakpoint	Do one of the following: • Select the line that contains the breakpoint, and then click **Debug** ▸ **Breakpoints** ▸ **Disable**. • Select the line that contains the breakpoint, and then press **Space**. • Right-click the line that contains the breakpoint, and then click **Disable Breakpoint**.
Enable all breakpoints	Click **Debug** ▸ **Breakpoints** ▸ **Enable/ Disable All Breakpoints**.

You can also

Enable a single line-number breakpoint	Do one of the following: • Select the line that contains the breakpoint, and then click **Debug ▸ Breakpoints ▸ Enable**. • Select the line that contains the breakpoint, and then press **Space**. • Right-click the line that contains the breakpoint, and then click **Enable Breakpoint**.

To log events for the breakpoints in a macro

1 While debugging a macro, do any of the following:
- Click **Debug ▸ Breakpoints ▸ Edit**.
- Right-click any line in the macro, and then click **Edit Breakpoints**.
- Press **Ctrl + B**.

2 Select a breakpoint, and on the **Actions** page, do any of the following:
- Enable the **Break into debugger** check box if you want that breakpoint to cause a break in the macro.
- Enable the **Log standard event message** check box if you want that breakpoint to log a standard event message.
- Enable the **Log custom message** check box, and specify the desired message, if you want that breakpoint to log a custom message.

3 Repeat step 2, as desired.

4 Click **Event Log**.

5 Enable the **Logging enabled** check box.

6 Click **Save**, and then specify a path and filename for the event log.

By default, the event log is saved with a **.log** extension.

To execute a token at a breakpoint in a macro

1 While debugging a macro, do one of the following:
- Click **View ▸ Execute Token**.
- Press **Alt + 0**.

2 Select the PerfectScript token that you want to execute, and then specify any parameter and return values that you want to use.

3 Click **Execute** to perform the command and display any return value.

Be careful when executing tokens. Some PerfectScript commands cause the internal state of the running macro to change and can therefore cause errors to occur later in the macro. (Most of these commands do not appear in the command list and cannot be selected.)

Working with variables while debugging macros

As explained in “Understanding variables” on page 10, a variable stores a value that can change during the operation of a macro.

When the Debugger stops at a breakpoint in a macro (see “Working with breakpoints while debugging macros” on page 150), the **Variables** list is updated to display, by name, the variables that are accessible to the macro at that location.

For each variable, the **Variables** list provides the pool type (**Local**, **Global** or **Persistent**), value type, and contents. For more information, see “Using the Variables list in the Debugger” on page 138.

While debugging a macro, you can use the **Variables** list to display, watch, and create variables.

Displaying variables in macros

You can use the **Variables** list to display all variables, or to display a combination of watched variables, local variables, global variables, and persistent variables.

The **Variables** list displays all variables that are defined at the current step in the macro. As you click the entries in the **Label, Function, Procedure** list, the variables in the list change to display the local variables that are defined in each step.

Even though the **Variables** list may display multiple variables with the same name, only the most locally scoped variable with that name is accessible to the macro as it executes. For example, if there is both a local variable and a global variable named `B`, only the local `B` can be accessed by the macro.

After a variable has been declared with the `DECLARE`, `LOCAL`, `GLOBAL`, or `PERSIST` statement in a macro, that variable appears in the **Variables** list even though its contents may be undefined.

When a macro contains a large number of variables, the **Variables** list initially displays only the first 100 variables. (In the case of an array, the **Variables** list displays only the first 100 elements, and of those, the values of only the first 25 elements.) If you want, you can expand or collapse the list of variables.

For more information, see "Displaying variables in macros" on page 156.

Watching variables in macros

If you are interested only in certain macro variables, you can add those variables to the **Watch** list. When the **Watch** list is displayed, it replaces the normal **Variables** list.

Only entire arrays or non-array variables can be watched. Individual array elements cannot be watched separately from their corresponding parent array.

For more information, see "To watch a variable while debugging a macro" on page 158.

Creating variables in macros

You can add variables to the macro that you are debugging. You can create new variables in any variable pool (**Local**, **Global**, or **Persistent**). Variables are created with undefined contents, but you can supply the value at any time.

For more information, see "To create a variable while debugging a macro" on page 158.

To display variables while debugging a macro

- Click **Variables** ▸ **View**, and then click any of the following:
  - **All** — displays all variables
  - **Watch List** — displays only watched variables
  - **Locals** — displays only local variables
  - **Globals** — displays only global variables
  - **Persistents** — displays only persistent variables.

  A checkmark next to a command indicates that that type of variable is displayed.

You can also

Sort the **Variables** list	Click a column heading, or click **Variables** ▸ **Sort By** and choose a sorting method. **TIP:** You can change the sort order of the **Variables** list from ascending to descending (or from descending to ascending) by clicking the heading of the sorting column.

You can also

Expand collapsed variables or a collapsed array in the **Variables** list	Select the collapsed item, and then click **Variables ▸ Expand**. **NOTE:** Although you can expand the contents of an array, doing so can consume a great deal of time and memory, so a warning prompts whether to expand all elements or just the first 100 elements. **TIP:** When only the first 100 elements of an array are displayed, a 101st element represents the remaining elements in the array but without their values. You can display the remaining elements by clicking the **{…}** icon next to the 101st element, or by collapsing the array and expanding it again (to display the warning that prompts whether to expand all elements).
Collapse expanded variables or an expanded array in the **Variables** list	Select the expanded item, and then click **Variables ▸ Collapse**.
Refresh the **Variables** list	Click **Variables ▸ Refresh**.

To watch a variable while debugging a macro

- Do one of the following:
  - Select the macro, and then click **Variables ▸ Watch**.
  - Right-click the variable in the **Variables** list, and then click **Watch**.

After a variable is added to the **Watch** list, it is displayed until you remove it from the list, even if the variable ceases to exist (for example, if it is removed in the macro). When a variable ceases to exist, its pool type is displayed as “out of scope.”

To create a variable while debugging a macro

1 When the Debugger stops at the desired breakpoint, click **Variables ▸ New**.

The **Create New Variable** dialog box appears.

2 Type a name for the variable, enable the option that corresponds to the desired variable type, and then click **Create**.

You can create an array variable by specifying the dimensions of the array after its name.

You can also

Edit a variable while debugging a macro	Select the variable (or select the array that contains the variable, and then select the variable), change the contents of the variable in the **Contents** box, and then press **Enter**. **NOTE:** You cannot change the contents of an alias variable, but you can change the variable that is mapped to the alias. **NOTE:** If you change the contents of an array, the new value is placed in all of its array elements. However, changing the contents of an array element changes that element only and does not affect the contents of any other array elements. **TIP:** You can cancel a change by pressing **Esc**.
Remove a variable while debugging a macro	Select the variable, click **Variables ▸ Delete**, and then click the desired discard option. (You can remove the variable from the **Variables** list box, or you can remove the contents of the variable but leave it in the **Variables** list box.) **NOTE:** Discarding an array variable first resets the contents of all the array elements. **TIP:** Be careful when discarding variables, in case the macro requires the variable to be defined.

Navigating the code while debugging macros

While debugging a macro, you have several options for navigating the code.

As previously discussed (see “To debug a macro by using the Debugger” on page 142), you can play through a macro during debugging. You can also pause, restart, or stop a macro during debugging.

Also as previously discussed (see “Working with breakpoints while debugging macros” on page 150), you can use breakpoints if you want to suspend debugging in places where you want to examine the macro more closely.

However, if you prefer, you can navigate macros by stepping through the code, using cursor position, enabling the animation feature.

You can also search the macro code if you want to locate a specific item.

Stepping through macros

You can go through a macro step-by-step. This process is called “stepping through the macro.”

The Debugger provides three commands for stepping through a macro:

- **Step Into** — executes the next single statement. If the next statement is a label call or a routine call, then execution steps into the specified label or routine (even if that label or routine is in another macro file, such as a **Use** file).
- **Step Over** — executes the call of the label or routine without stopping until its completion. The macro stops at the next statement in the current label or routine.
- **Step Out** — executes until the next return is encountered (if a label or routine is entered)

For information about this procedure, see “To step through a macro in the Debugger” on page 161.

Using cursor position in macros

The Debugger provides two commands for going through a macro by using cursor position:

- **Run to Cursor** — continues execution down to the line under the mouse cursor in the **Source** list
- **Skip to Cursor** — sets the next statement to be executed by the Debugger, without executing any statements between the current point and the new line. This feature lets you skip a series of statements without executing them, or allows for statements to be repeated.

For information about this procedure, see “To use cursor position while debugging a macro” on page 161.

Animating macros

The Debugger lets you animate macros. Animation allows you to step through macros line-by-line, automatically activating the Debugger at each step so that you can check the variables and other calls. Between each command, the Debugger is displayed for the amount of time specified in the **Tools ▸ Settings** dialog box.

For information about this procedure, see "To animate a macro in the Debugger" on page 161.

Searching macros

The Debugger lets you search a macro for a specific line number or text string. The Debugger also lets you locate the current line of code when a macro is interrupted.

For information about this procedure, see "To search macro code in the Debugger" on page 162.

To step through a macro in the Debugger

- While debugging a macro, do any of the following:
  - Click **Debug ▸ Step Into** to execute the next single statement.
  - Click **Debug ▸ Step Over** to execute the call of the label or routine without stopping until its completion.
  - Click **Debug ▸ Step Out** to execute the macro until the next return is encountered (if a label or routine is entered).

To use cursor position while debugging a macro

1 While debugging a macro, position the cursor by clicking in the **Source** list.

2 Do one of the following:
  - Click **Debug ▸ Run to Cursor** to continue execution down to the line under the mouse cursor in the **Source** list.
  - Click **Debug ▸ Skip to Cursor** to set the next statement to be executed by the Debugger, without executing any statements between the current point and the new line.

You can use the **Skip to Cursor** command if you need to skip a series of statements without executing them, or if some statements need to be repeated. However, this feature must be used with extreme caution: Skipping to a line that is not within the same label or routine can cause the internal macro execution state to become invalid and result in execution failure.

To animate a macro in the Debugger

- In the Debugger, click **Debug ▸ Animate** to begin playing the macro by using the specified animation settings.

If the macro pauses, click **Debug ▸ Animate** to resume automatic play.

For information about specifying the Debugger settings for animation and other debugging features, see "Setting up the Debugger" on page 139.

To search macro code in the Debugger

- While debugging a macro, do any of the following:
  - To search for a specific line number, click **Edit ▸ Find Line Number** (or press **Ctrl + G**), and then specify the line number.
  - To search for specific text, click **Edit ▸ Find Text** (or press **Ctrl + F**), and then specify the text and the search options in the **Find Text in Source** dialog box that appears.
  - To search for the next instance of text that is specified in the **Find Text in Source** dialog box, click **Find Next** (or press **F3**).
  - To search for the previous instance of text that is specified in the **Find Text in Source** dialog box, click **Find Previous** (or press **Shift + F3**).

For information about searching for breakpoints, see "To move between the breakpoints in a macro" on page 154.

You can also

Locate the current line of code when a macro is interrupted	Click **Edit ▸ Find Current Line**. The entire line of code is highlighted in the **Source** list.

Troubleshooting the Debugger

The techniques that you use to debug a macro depend on the type of failure that is occurring. This section describes two such types of issues:

- macros that stop at error messages
- macros that contain faulty callbacks

Debugging macros that stop at error messages

One of the easiest problems to correct with the Debugger is when a macro terminates due to an error.

To discover the cause of such an error, use the Debugger to play the macro as usual. When the macro stops, click **Debug** on the displayed error-message box to invoke the Debugger and load both the macro and the listing file for the macro compiler. The Debugger displays the following:

- the problem line in the macro (see the **Source** list)
- in reverse order, the labels, functions, and procedures that were called to get the macro to that point (see the **Call History** list)
- the contents of the variables in the macro (see the **Variables** list)

At this point, you can examine the source line to determine what the macro was doing when the error occurred. You can also check the contents of the variables that are being used at the current line to see if they contain incorrect, error-causing values.

If all the variables at this point in the macro appear to have the correct values, select a previous line in the **Call History** list, and examine the source line and the contents of the variables at that point. If you find a variable value that seems improper, examine the source code that leads to the problem area. Locate a spot where the variable could have been changed, and double-click the line to set a line-number breakpoint at that spot.

Now stop the Debugger by clicking **Debug ▸ Stop Debugging**. You cannot continue executing the macro because an error has occurred.

Restart the macro by clicking **File ▸ Debug ▸ Play** in PerfectScript. This displays the Debugger immediately before the macro starts.

Because you set a line-number breakpoint, click **Continue** and run the macro until it encounters the line number that contains that breakpoint.

When the macro reaches the line with the breakpoint, the Debugger is displayed. Examine the variable contents to verify that the macro is displaying the variable values that you expect. If not, you may need to specify a breakpoint at an earlier location and then restart the macro. You can also skip to the earlier location by clicking a line in the macro source and then clicking **Debug ▸ Skip to Cursor**.

If everything appears to be working normally, click **Step Into** to execute the macro one statement at a time. At each step, examine the variables between each statement until something unexpected happens. For example, the contents of a variable may change unexpectedly, or the macro may call the wrong label. At this point, you have probably come to the line where the value of the variable causes the error.

If it appears that the error occurs because a variable has the wrong contents, set a "variable assign" breakpoint for that variable by clicking **Debug ▸ Breakpoints ▸ Edit** and then choosing **Variable Assign** from the **Type** list box. When the contents of that variable change, a breakpoint occurs and the Debugger is displayed. If the variable is changed too often and too many breakpoints are occurring on the variable, disable the **Variable Assign** breakpoint and set a line-number breakpoint after most of the variable assignments are done but before the wrong contents are assigned. If the problem in the macro appears to associated with calling a certain product token, set a **Product Token Call** breakpoint for that product token, or for a related product token that may not be setting the application in the proper state, or for both. If the problem is that the macro is calling a specific label, function, or procedure too often, set a **Label Call** breakpoint for that label to see when and where it is being called.

Debugging macros that contain faulty dialog-box callbacks

Debugging dialog-box callbacks is one of the most challenging types of debugging. Many problems with dialog-box callbacks occur because the callback does not reference the callback data closely enough to determine if the exact conditions have occurred before performing the action; consequently, the action is performed at the wrong time (or too many times).

A dialog-box callback label is called often, and most of the callbacks are usually not associated with the event of interest. As soon as a callback dialog box is shown, at least two or three callback events occur. These events are associated with the creation of the dialog box, the initialization of the controls, and the initial input focus that is received by the dialog box.

One of the difficulties that are associated with debugging callbacks is that when the Debugger becomes active at a breakpoint, the Debugger receives focus over any window that had focus while the macro was running. Therefore, the callback dialog box in the macro also loses focus. When a callback dialog box loses or gains focus, a callback event is generated.

If you set a breakpoint in a dialog-box callback label, a lose-focus callback event could occur on the callback dialog box when the Debugger is displayed. Then, when you continue macro execution in the Debugger, a gain-focus callback event could occur. This scenario could cause the breakpoint to stop the Debugger (because the callback label is called for the lose/gain focus of the dialog box), which would, in turn, cause more lose/gain callback events, which could cause another breakpoint, which could cause more lose/gain focus callback events, and so on.

The macro interpreter and the Debugger attempt to minimize this cyclic occurrence by determining whether the dialog box is losing or gaining focus because of the Debugger. If so, the focus callback events are not generated. Unfortunately, this explanation cannot always be reliably determined.

The best way to prevent error loops is to write the dialog-box callback label so that it carefully examines the callback data array to determine the type of the callback event, and to have it respond only to specific callback event types. (Callbacks usually respond to events that are associated with manipulating a control, which are `WM_COMMAND` events with element `[5] = 273`). Next, set a breakpoint on a line of code that does not respond to losing or gaining focus. In this way, the breakpoint does not occur when the dialog box loses or gains focus, circumventing the error cycle.

If a callback event occurs for a callback label that is different from any callback label that is currently executing, then the callback event causes the current callback code to be suspended, and the new callback code is called. By restricting this focus to different callback labels from any currently active callback label, you can ensure that a callback can be completed before another callback for that same callback label is allowed to occur.

For more information, see “Setting up callbacks for dialog boxes” on page 124.

For more information about dialog boxes, see “Creating dialog boxes for macros” on page 101.

Glossary

A

ANSI character set

The 256 characters of the American National Standards Institute

any data type

A data type that accepts more than one data type as input. For example, in the `AppActivate (Windows: Any)` command, the `Windows` parameter accepts a window title (string data type) or a window handle (numeric data type).

argument

A variable, constant, or expression required by a command or function

B

Boolean data type

A data type that accepts or returns a value of `True` or `False`

C

callback

A special function that enables a macro to respond immediately and in specific ways to events, such as enabling a radio button or check box, without waiting until a dialog box is dismissed

character expression

Also called a string, one or more characters enclosed in quotation marks. This syntax identifies the characters as text, rather than as a variable.

Command Browser

Also called the Command Inserter, a dialog box that inserts product commands or programming commands (or both) into a macro document

command name

A description of a command's action, such as `Font`, `MarginLeft`, `Advance`, or `FootnoteOptions`

Command names can execute product features ("product commands") or direct the execution of the macro ("programming commands," such as `If`, `Else`, or `End If`). The Command Inserter lists programming commands and product commands separately, but you can choose which list you want to display.

constant

A named item that keeps a constant value while a macro is being executed

control statement

A macro feature that alters the sequential execution of commands

D

data type

The set of values that a variable can store. A data type represents information that is needed by a parameter or returned by a command (as a "return value").

The available data types are Boolean, enumeration, label, measurement, numeric, string, variant, and any.

In the command syntax, data types are displayed in italics. For example, the enumerations for the `Rotation` parameter of `BoxCaptionRotation` are `Degrees90!`, `Degrees 180!`, `Degrees 270!`, and `None!`. Only these enumerations can replace the data type in the command syntax.

DLL (Dynamic Link Library) file

A library of functions and procedures that can be called from a macro

drop-down list

A type of list available to a Combo Box control. Also called a "list box."

E

enumeration

An option provided by the program, such as a style, type, method, or state. Also called an "enumerated type."

Enumerations end with an exclamation point.

For example, the `DisplayMode` parameter accepts only `Text!`, `Graphics!`, or `FullPage!` as an emueration. In WordPerfect, `On!`, `Heading8Style!`, and `DefFlushRight!` are enumerations used by different commands.

enumeration data type

A data type that accepts an enumeration

event

A noun that acts as something taking place in an object and that is triggered by an action (such as a click, key press, or system timer)

event-driven programming

A form of programming, such as Visual Basic for Applications, in which code is executed in response to events. By contrast, in traditional procedural programming, the program starts at line 1 and executes line by line.

expression

An element that represents values. An expression can be arithmetic, numeric, measurement, relational, logical, bitwise, or character (that is, a string).

L

label

A subroutine similar to a procedure or function. A label generally contains one or more statements followed by `Return` or `Quit`.

label data type

A data type that accepts a label

M

Macro toolbar

A toolbar that contains tools for writing and editing macros. It features buttons for saving, compiling, inserting macro commands, and so on.

measurement data type

A data type that accepts a measurement value in inches, millimeters, picas, WP units, and so on. For example, `72P` (points) is equal to `1I` (inch) and to `2.54C` (centimeters).

All measurement return values are returned in WordPerfect units. Necessary unit conversions are done internally when comparing two measurement values. Recorded macros use the units specified in the application preferences. When specifying a measurement value in a product command parameter, WordPerfect units (`W`) are assumed unless other units are specified in the parameter or with the `DefaultUnits` command.

measurement expression

A number followed by a unit of measure (`"`, `i`, `c`, `m`, `p`, `w`)

N

numeric data type

A data type that accepts a numeric expression

numeric expression

Also called a "numeric," a number (such as the number of seconds, a line number, or an outline level) on which mathematical operations can be performed. Numeric expressions are not enclosed in quotation marks.

O

object model

The hierarchy of objects within an application and their relationship to each other within the paradigm. Each object within an object model is defined by a property, method, or event — or by a combination of each. An object responds to an action through the use of written code.

For example, the `Document` object represents the beginning of the object hierarchy in WordPerfect. Starting with the `Document` object, you can drill down and navigate through the object model until you find the desired object. To reference an object with Visual Basic code, you can separate each level of the object hierarchy with the dot operator (`.`).

object-oriented programming

A form of programming that emphasizes creating and using objects

OLE (Object Linking and Embedding)

A feature that copies information from one document to another, "embedding" it through a "live" link. When the original document changes, the embedded copy reflects the changes.

OLE object command

An item, also called a "method," that performs tasks on an OLE object in a specific OLE automation server.

OLE object commands are specific for each object, such as `Excel.Application` or `Excel.Workbooks`. They perform various functions on that object.

OLE object commands that return information about an object are called "properties." Many properties have parameters as well as return values. In addition, many properties can be assigned a value by placing them on the left side of the assignment symbol (`:=`), similarly to the name of a variable.

operator

A symbol or word that performs a function on one or more expressions. Operators compare expressions, link words together, and perform mathematical functions.

P

parameter

An optional command element. For example, `InhibitInput (State: Off!)` works just the same as `InhibitInput (Off!)`.

Some product commands have no parameters; their syntax is usually written with empty parameters, such as `PosScreenUp()`. Similarly, some programming commands and WordPerfect system variables have no parameters; their syntax is the command name alone, such as `PAUSE` and `?FeatureBar`.

In this documentation, italics indicate parameter names or types to be replaced with data. For example, the syntax of `GraphicsLineLength` is as follows:

```
GraphicsLineLength (Length: measurement)
```

After you replace `measurement` with a number, the command might be

```
GraphicsLineLength (Length: 2I)
```

or

```
GraphicsLineLength (2I)
```

Be sure to enclose parameters in parentheses. A missing parenthesis, either opening or closing, is a common error that prevents macros from compiling.

Spaces between command names and the opening parenthesis of the parameter section and after semicolons in parameters are optional.

You can separate multiple parameters with semicolons (`;`). If you omit an optional parameter, be sure to include the semicolon in the syntax to keep following parameters in their correct positions, as in this example:

```
AbbreviationExpand (AbbreviationName:; Template: PersonalLibrary!)
```

or

```
AbbreviationExpand (; PersonalLibrary!)
```

You can enclose repeating parameters in braces, as in this example:

```
CASE (<Test>: Any; { <Case>: Any; <Label>: label; Case: Any; <Label>:
<Label>...}. When data is supplied, the command could be CASE
(vChoice; { 1; Exclaim; 2; Info; 3; Question; 4; Stop; 5; QuitMacro};
QuitMacro)
```

PerfectScript

An application used to record, play, compile, convert, and edit macros. PerfectScript is used to build or edit dialog boxes for macros.

product command

A command name that is specific to each application (such as WordPerfect or Presentations) and that performs various functions in that application

For example, product commands (such as `InitialCodesStyleDlg`) can display a dialog box; specify settings such as styles (`BorderBottomLine`), user preferences (`PrefZoom`), or attributes (`Font`); turn features on and off (`InhibitInput` or `TableCellIgnoreCalculation`); perform actions such as inserting a file (`FileInsert`) or code (`PrinterCommand`), renaming a bookmark (`BookmarkRename`), converting comments to text (`CommentConvert`), or moving the insertion point (`PosColBottom`); or play macros that are included with the application (`FontDnShippingMacro`).

Product commands that report information about the state of an application or feature (that is, provide a “return value”) are sometimes called “system variables.” In WordPerfect, system variables begin with a question mark (for example, `?ColumnWidth`). In Presentations, they begin with `Env` (for example, `EnvPaths`). Some system variables in Presentations have parameters as well as return values.

The most common data types in product commands are string, enumeration, and numeric.

product prefix

A two-character expression that specifies a product for a macro command

programming command

A command name that works across applications and that controls or returns information about applications and feature functions.

Programming commands generally control macro functions, such as by specifying conditions under which other macro commands or statements operate (`CASE`, `IF ELSE ENDIF`, `SWITCH ENDSWITCH`); repeating macro commands or statements a specified number of times or until certain conditions are met (`FOR ENFOR`, `REPEAT UNTIL`, `WHILE ENDWHILE`); or invoking or jumping to a specified subroutine (“statement block”) with `CALL` or `GO`.

Programming commands are frequently variables.

prompt

A dialog box box that displays information for the user

R

relational expression

An expression that evaluates parameters with only two possible states: `TRUE` and `FALSE`

run-time

The period during which a macro is executed. Run-time errors occur during macro execution. Run-time options are application start-up settings, such as the macro’s default directory.

S

string

Also called a character expression, one or more characters enclosed in double quotation marks. This syntax identifies the characters as text, rather than as a variable.

Strings can include numbers. Here is a sample string:

```
Type ("1") and MESSAGEBOX (vStatus; "VAR" + x; "Continue to next
variable?" ; IconQuestion! |YesNo!!)
```

string data type

A data type that accepts a string

The string data type represents text that you provide, such as a filename, a dialog box control name, message box text, or a character sequence to insert into a document.

syntax

The grammar or sequence for assembling commands

T

toggle command

A command that switches between states

For example, the WordPerfect `Bold` command can be `On!` or `Off!`. If `On!` or `Off!` is not specified, `Bold` "toggles" between the two states each time it is called; if `Bold` is `Off!`, it is turned `On!`, while if `Bold` is `On!`, it is turned `Off!`.

token ID

The name of a macro command (such as `InvokeDialog`)

U

user-defined dialog box

A custom dialog box created with `Dialog` programming commands that display options for user input

V

value set member

A value in an enumeration list

variable

An item that can be created (or "declared") for the purposes of storing data

A variable name must begin with a letter but is not case-sensitive. (All variables in this documentation's examples begin with `v`.) Variables can include any combination of letters or numbers up to 50 characters. For example, you could "assign" the value `C:\COREL\WPO\` to variable `vPath`, and then substitute the variable for the path in the rest of the macro.

You can change the value of a variable. If variable `vNmbr` equals 5, then the expression `vNmbr := vNmbr + 1` results in `vNmbr` equaling 6.

variant data type

The data type for all variables that are not declared as another type (such as `Dim`, `Private`, `Public`, or `Static`). The variant data type has no type-declaration character.

W

window

The application area that contains a title bar, menu bar, and application bar, and that may contain a property bar, scroll bar, toolbar, and ruler. The **Equation Editor** window has separate editing and display areas called "panes."

window handle

A unique identifier for a window or control

Index