POWERSOFT
ENTERPRISE
SERIES

PowerScript Language

VERSION 4.0

PowerBuilder

December 1994

Contents

About This Manual

Subject

This manual describes the PowerScript language, which is the language you use in scripts and user-defined functions to build PowerBuilder applications.

Audience

This manual is for programmers who will be building and maintaining PowerBuilder applications. It assumes that you are familiar with Microsoft Windows 3.1 and the SQL statements supported by your database management system (DBMS).

CHAPTER 1

Language Basics

About this chapter

This chapter describes general elements and conventions of PowerScript.

Contents

Comments

You can use comments to document your scripts and to prevent statements within a script from executing.

There are two ways to designate comments in PowerScript: the **double slash** method and the **slash and asterisk** method.

> **Tip**
> In the PowerScript painter and the Function painter, you can use the Comment Selection button or select Edit➤Comment Selection from the menu bar to comment out the line containing the cursor or a selected group of lines.

For information about adding comments to objects and library entries, see the *User's Guide*.

Double slash method

You use the double slash method to designate a single line comment. The comment can be the entire line or part of the line. When the compiler encounters double slashes, it ignores everything following double slashes and on the same line. When you use this method to designate a comment, the comment *cannot* extend to multiple lines.

Examples

The following examples show how to use the double slash method to designate comments.

```
// This entire line is a comment.
// This entire line is another comment.

amt = qty * cost   // Rest of the line is comment.

// The following statement was commented out so it
// would not execute.
// SetNull(amt)
```

Slash and asterisk method

With the slash and asterisk method, a slash followed by an asterisk (/*) begins a comment and an asterisk followed by a slash (*/) ends the comment. The compiler ignores everything between the slash asterisk and the asterisk slash. When you use this method to designate a comment, you can:

- Make all or part of a line a comment
- Extend a comment to multiple lines
- Nest comments

Continuing comments
Multiline comments do not require a continuation character.

Examples

```
/* This is a single-line comment.   */

/* This comment starts here,
continues to this line,
and finally ends here. */

A = B + C  /*  This comment starts here.
/*  This is the start of a nested comment.
    The nested comment ends here.  */
The first comment ends here.  */ + D + E + F
```

Summary

Delimiter	Use to
//	Designate all or part of a line as a comment
/*...*/	Designate all or part of a line as a comment or multiple lines as a single comment Nest comments

Identifier names

You use identifiers to name variables, labels, functions, windows, controls, menus, and anything else you refer to in scripts.

Rules

Identifiers:

- Must start with a letter
- Can have up to 40 characters, but no spaces
- Are case insensitive (PART, Part, and part are identical)
- Can include any combination of letters, numbers, and these special characters:

-	Dash
_	Underscore
$	Dollar sign
#	Number sign
%	Percent sign

Prohibiting dashes in variable names

By default, PowerBuilder allows you to use dashes in all identifiers, including in variable names in a script. This means that when you use the subtraction operator or the -- operator in a script, you must surround it with spaces (otherwise, PowerBuilder thinks the expression is an identifier name).

If you want to disallow dashes in variable names in scripts (and not have to surround the subtraction operator and -- with spaces), you can set the DashesInIdentifiers preferences variable to 0 in the [pb] section of PB.INI.

```
[pb]
DashesInIdentifiers=0
```

By default, DashesInIdentifiers equals 1, which allows dashes.

Changing DashesInIdentifiers
Be careful: if you do set the variable to 0 and have previously used dashes in variable names, you will get errors the next time you compile.

Using multiword names

Since PowerScript does not allow spaces in identifier names, you can use any of the following techniques for multiword names.

- Initial caps (for example, FirstWindow)
- Dashes, except in variable names if you set DashesInIdentifiers to 0 (for example, customer-name)
- Underscores (for example, quantity_on_hand)

Examples

Here are some valid identifiers.

```
ABC_Code
Child-Id
FirstButton
response35
pay-before%deductions$
ORDER_DATE
Actual-$-amount
Part#
```

Here are some invalid identifiers.

```
2nd-quantity // Does not start with a letter
ABC Code     // Contains a space
Child'sId    // Contains invalid special character
```

Labels

You can include labels in scripts for use with GOTO statements. A label can be any valid identifier followed by a colon (:). You can enter it on a line by itself or at the start of the line preceding a statement.

👓 For information about the GOTO statement, see Chapter 5, "Statements."

Examples

The label shown below is on its own line and above the statement.

```
FindCity:
IF city=cityname[1] THEN ...
```

The label shown below is on same line as the statement.

```
FindCity:   IF city=cityname[1] THEN ...
```

ASCII characters

You can include special ASCII characters in strings. For example, you may want to include a tab in a string to ensure proper spacing or a bullet to indicate a list item. The tilde character introduces special characters.

Common ASCII characters

To specify this ASCII character	Enter
Newline	~n
Tab	~t
Vertical tab	~v
Carriage return	~r
Formfeed	~f
Backspace	~b
Double quote	~"
Single quote	~'
Tilde	~~

The following table illustrates how to use special characters in strings.

String	Description
"dog~n"	A string containing the word dog followed by a newline character
"dog~tcat~ttiger"	A string containing the word dog, a tab character, the word cat, another tab character, and the word tiger

ASCII values

You can specify *any* ASCII character (including the characters in the previous table) by typing a tilde (~) followed by the decimal, hexadecimal, or octal ASCII value for the character.

ASCII value	Enter
Decimal	A tilde followed by three digits from 000 to 255
Hexadecimal	A tilde followed by a lowercase h, followed by a two-digit hexadecimal number from 01 to FF
Octal	A tilde followed by a lowercase o, followed by a three-digit octal number from 000 to 377

Examples

The following table shows how to indicate a bullet (•) in a string by using the decimal, hexadecimal, and octal ASCII values.

Value	Description
~249	The ASCII character with decimal value 249
~hF9	The ASCII character with hexadecimal value F9
~o371	The ASCII character with octal value 371

NULL values

Although PowerBuilder supports NULL values for all variable data types, it does *not* initialize variables to NULL. Instead, when a variable is not set to a specific value when it is declared, PowerBuilder sets it to the default initial value for the data type. For example, zero for a numeric value, FALSE for boolean, and the empty string ("") for a string.

Typically, you work with NULL values only with respect to database values.

What NULL means

NULL means undefined. Think of NULL as unknown. It is not the same as an empty string or zero or a date of 0000-00-00. For example, NULL is neither 0 nor not 0.

NULL variables

A variable can become NULL if one of the following occurs:

- A NULL value is read into it from the database. If your database supports NULL and a SQL INSERT or UPDATE statement sends a NULL to the database, it is written to the database as NULL and can be read into a variable by a SELECT or FETCH statement.

  Tip
  When a NULL value is read into a variable, the variable remains NULL unless it is changed in a script.

- The SetNull function is used in a script to set the variable explicitly to NULL. For example:

  ```
  string  city  // city is an empty string.
  SetNull(city) // city is set to NULL.
  ```

NULLs in functions and expressions

Any function that has a NULL value for *any* argument returns NULL. Any expression that has a NULL variable results in NULL.

Examples

None of the following statements will make the computer beep. The variable nbr is set to NULL, so each statement evaluates to NOT TRUE.

```
int   Nbr
// Set Nbr to NULL.
SetNull(Nbr)
IF Nbr =1 THEN Beep(1)
IF Nbr <> 1 THEN Beep(1)
IF NOT (Nbr = 1) THEN Beep(1)
```

In the following IF...THEN statement, the expression evaluates to NOT TRUE, so the ELSE is executed.

```
int   a
SetNull(a)
IF a = 1 THEN
   MessageBox("Value", "a = 1")
ELSE
   MessageBox("Value", "a = NULL")
END IF
```

This is very useful. For example, the following statement displays a message if no control has focus (if no control has focus, GetFocus returns a null object reference).

```
IF GetFocus( ) THEN
   . . .  // Some processing
ELSE
   MessageBox("Important", "Specify an option!")
END IF
```

Testing for NULL

To test whether a variable or expression is NULL, use the IsNull function. You *cannot* use an equal sign (=) to test for NULL.

Examples

These statements show the correct and incorrect way to test for NULL.

```
IF IsNull(a) THEN ... // This is correct.

IF a = NULL THEN  ... // This is not valid.
```

Reserved words

The words PowerBuilder uses internally are called reserved words and generally cannot be used as identifiers. The exceptions are *Parent*, *This*, *ParentWindow*, and *Super*. You can use these pronouns to make general references in scripts to objects and controls.

👓 For a list of PowerBuilder reserved words, see Appendix B, "Reserved Words."

Parent, This, ParentWindow, and Super

When you use Parent, This, ParentWindow, or Super to make a general reference to an object or control, the reference is correct even if the name of the object or control changes.

You can use these pronouns in functions to cause an event in an object or control, or to manipulate or change an object or control. You can also use these pronouns to obtain or change the setting of an attribute.

Each of these pronouns has a specific meaning and use.

Parent

You can use the pronoun Parent in the following scripts:

- Scripts for a control in a window
- Scripts for a custom user object
- Scripts for a MenuItem

Where you use Parent determines what it references.

Window controls

When you use Parent in a script for a control (such as a CommandButton), Parent refers to the window that contains the control. For example, if you include the following statement in the script for the Clicked event in a CommandButton within a window, clicking the button closes the window containing the button:

```
Close(Parent)
```

If you include the following statement in the script for the CommandButton, clicking the button displays a horizontal scrollbar within the window (sets the HScrollBar attribute of the window to TRUE).

```
Parent.HScrollBar = TRUE
```

User object controls

When you use Parent in a script for a control in a custom user object, Parent refers to the user object. For example, if you include the following statement in a script for the Clicked event for a CheckBox in a user object, clicking the checkbox hides the user object.

```
Parent.Hide( )
```

If you include the following statement in the script for the CheckBox, clicking the checkbox disables the user object (sets the Enabled attribute of the user object to FALSE).

```
Parent.Enabled = FALSE
```

MenuItems

When you use Parent in the script for a MenuItem, Parent refers to the MenuItem on the level above the MenuItem the script is for. For example, if you include the following statement in the script for the Clicked event in the MenuItem Select All under the MenuItem Select, clicking Select All disables the MenuItem Select.

```
Parent.Disable( )
```

If you include the following statement in the script for the Clicked event in the MenuItem Select All, clicking Select All checks the MenuItem Select.

```
Parent.Checked = TRUE
```

This

The pronoun This refers to the window, user object, MenuItem, application object, or control itself.

Examples

For example, if you include the following statement in the script for the Clicked event for a CommandButton, clicking the button changes the horizontal position of the button (changes the button's X attribute).

```
This.X = This.X + 50
```

Similarly, the following statement in a script for a MenuItem places a checkmark next to the MenuItem.

```
This.Check( )
```

Why include This

In the script for an object or control, you can refer to the attributes of the object or control without qualification. However, it is good programming practice to include This to make the script easy to read and to add clarification.

For example, if you omit This in the statement shown above, the statement accomplishes the same result but looks like this.

```
x = x + 50
```

However, if you omit This and there is a variable named x within the scope of the script, the variable takes precedence (the script adds 50 to the variable x, not to the X attribute of the control).

Also, you can use This in a function call to pass a reference to the object containing the script, such as:

```
ReCalc(This)
```

ParentWindow

The pronoun ParentWindow refers to the window that a menu is associated with at execution time. ParentWindow can be used only in scripts for MenuItems.

Examples

For example, the following statement in a script for a MenuItem closes the window the menu is associated with at execution time.

```
Close(ParentWindow)
```

This statement in the script for a MenuItem reduces the height of the window the menu is associated with at execution time.

```
ParentWindow.Height = ParentWindow.Height/2
```

However, the following statement in the script for a MenuItem is not valid. You cannot use ParentWindow to qualify a reference to a control.

```
ParentWindow.sle_Result.Text = ... // INVALID
```

Super

When you write a script for a descendant object or control, you can call scripts written for any ancestor. You can directly name the ancestor in the call, or you can use the reserved word Super to refer to the immediate ancestor (parent).

Examples

For example, to call the parent's Clicked script, code the following.

```
CALL Super::Clicked
```

Note that you can't use Super to call scripts associated with controls in the ancestor window.

If you are calling an ancestor function, you only need to use Super if the descendant has a function with the same name and the same arguments as the ancestor function. Otherwise, you would simply call the function with no qualifiers.

This example calls the ancestor function wf_myfunc. Presumably, the descendant also has a function called wf_myfunc.

```
Super::wf_myfunc()
```

You can only use Super in an event or function associated with a direct descendant of the ancestor who's function is being called. Otherwise the compiler will return a syntax error. The example above would have to be part of a script or function in the descendant window, not one of the window's controls. For example, if it were in the Clicked event of a button on the descendant window, you would get a syntax error when the script was compiled.

Summary

Reserved word	In a script for a	Refers to the
Parent	Control in a window	Window containing the control
	Control in a custom user object	Custom user object containing the control
	MenuItem	MenuItem on the level above the item the script is for
This	Window, custom user object, MenuItem, application object, or control	Object or control itself
ParentWindow	MenuItem	Window the MenuItem is associated with at execution time
Super	A descendant object or control	Parent

Statement continuation and separation

Although you typically put one statement on each line, you will occasionally want to continue a statement to more than one line or combine multiple statements on a single line.

Continuation character

The PowerScript continuation character is the ampersand (&). To continue a statement to another line, insert an ampersand wherever there is white space at the end of a line and then start the new line. The ampersand must be the last nonwhite character on the line (or the compiler will consider it part of the statement). White space is discussed at the end of this chapter.

Examples

This statement is continued across two lines.

```
IF Index = 3 AND &
   Count =4 THEN Beep(4)
```

This statement is continued across three lines.

```
Total-Cost = Price&
   * Quantity +&
   (Tax + Shipping)
```

Continuing a quoted string

You can continue a quoted string by simply placing an ampersand in the middle of the string and continuing the string on the next line.

```
IF Employee_District = "Eastern United State and&
Eastern Canada" THEN ...
```

Note that any white space (for example, tabs and spaces) before the ampersand and at the beginning of the continued line is part of the string.

To keep unwanted white space out of the string, a better way to continue a quoted string is to enter a quotation mark before the continuation character ('& or "&, depending on whether the string is delimited by single or double quotation marks) at the end of the first line of the string and a plus sign and a quotation mark (+' or +") at the start of the next line.

This method ensures that you do not inadvertently include unwanted characters, such as tabs or spaces, in the string literal. The examples in the PowerBuilder documentation and online Help use this method to continue quoted strings.

Examples

The following statement uses only the ampersand to continue the quoted string in the IF...THEN statement to another line. Note that a tab was used at the start of the second line to make the script easier to read.

```
IF Employee_District = "Eastern United States and&
   Eastern Canada" THEN ...
```

When you use the method shown above to continue the string, the compiler includes the tab in the string, which may result in an error. When you use the recommended method (shown below), the tab is not included in the string.

```
IF Employee_District = "Eastern United States and "&
   +" Eastern Canada" THEN ...
```

Continuing a variable name

You *cannot* split a line by inserting the continuation character within a variable name. This will cause an error.

Examples

The following statement will fail, because the continuation character splits the variable name (Quantity).

```
Total-Cost = Price * Quan&
   tity + (Tax + Shipping)
```

The following statement is valid, because "Price * Quantity + (Tax + Shipping)" is a quoted string, so Quantity can be split.

```
Total-Cost = "Price * Quan"&
   +"tity + (Tax + Shipping)"
```

Continuing a comment

Do not use a continuation character to continue a comment. The continuation character is considered part of the comment and is ignored by the compiler.

Continuing a SQL statement

Do not use a continuation character to continue a SQL statement. In PowerBuilder, SQL statements always end with a semicolon (;). The compiler considers everything from the start of a SQL statement until it encounters a semicolon to be part of the SQL statement. A continuation character in a SQL statement is considered part of the statement and usually causes an error.

Statement separator

The PowerScript statement separator is the semicolon (;). Use it to separate multiple statements *on a single line* to conserve space when there are a number of short, related statements in a script.

Example

The following line contains three short statements.

```
A = B + C;  D = E + F;  Count = Count + 1
```

White space

Blanks, tabs, formfeeds, and comments are forms of white space. The compiler ignores them unless they are part of a string literal (enclosed in single or double quotation marks).

Examples

In this example, the spaces and the comment in the expression are white space, so the compiler ignores them:

```
A + B /*Adjustment factor */+C
```

However, the spaces in the following expression are within a string literal, so the compiler does not ignore them.

```
"The value of A + B is:"
```

The subtraction operator

Unless you have prohibited the use of dashes in identifiers, you must surround the subtraction operator (minus sign) with spaces. If you don't, PowerBuilder will consider the operator part of a variable name:

```
Order - Balance   // Subtracts Balance from Order
Order-Balance     // A variable named Order-Balance
```

☞ For information on the use of dashes in names, see "Identifier names" on page 4.

CHAPTER 2

Data Types

About this chapter

This chapter describes the three kinds of data types provided by PowerScript.

Contents

Standard data types

The standard data types are the familiar data types that are used in many programming languages, including char, integer, decimal, long, and string. In PowerScript, you use these data types when you declare variables or arrays.

This section:

- Lists all standard PowerScript data types
- Describes the use of literals
- Describes the string and char data types

List of the standard data types

The following table lists all standard PowerScript data types.

Data type	Description
Blob	Binary large object. Used to store an unbounded amount of data (for example, generic binary, image, or large text, such as a word-processing document).
Boolean	Contains TRUE or FALSE.
Char or character	A single ASCII character.
Date	The date, including the full year (1000 to 3000), the number of the month (01 to 12), and the day (01 to 31).

Data type	Description
DateTime	The date and time in a single data type, used only for reading and writing DateTime values from and to a database. To convert DateTime values to data types that you can use in PowerBuilder, use: ♦ The Date(datetime) function to convert a datetime value to a PowerBuilder date value after reading from a database ♦ The Time(datetime) function to convert a datetime value to a PowerBuilder time value after reading from a database ♦ The DateTime (date, time) function to convert a date and (optional) time to a DateTime before writing to a DateTime column in a database PowerBuilder supports microseconds in the database interface for any DBMS that supports microseconds.
Decimal or Dec	Signed decimal numbers with up to 18 digits. You can place the decimal point anywhere within the 18 digits. For example, 123.456, 0.000000000000000001, or 12345678901234.5678.
Double	A signed floating-point number with 15 digits of precision and a range from 2.2E-308 to 1.7E+308.
Integer or Int	16-bit signed integers, from -32768 to +32767.
Long	32-bit signed integers, from -2,147,483,648 to +2,147,483,647.
Real	A signed floating-point number with six digits of precision and a range from 1.17 E -38 to 3.4 E +38.
String	Any ASCII characters with variable length (0 to 60,000).
Time	The time in 24-hour format, including the hour (00 to 23), minute (00 to 59), second (00 to 59), and fraction of second (up to six digits) with a range from 00:00:00 to 23:59:59.999999. PowerBuilder supports microseconds in the database interface for any DBMS that supports microseconds.
UnsignedInteger, UnsignedInt, or UInt	16-bit unsigned integers, from 0 to 65,535.
UnsignedLong or ULong	32-bit unsigned integers, from 0 to 4,294,967,295.

Using literals

You use literals to assign values to variables of the standard data types. PowerScript supports the following types of literals: date, decimal, integer, real, string, and time.

You use integer literals to assign values to data types that can contain only whole numbers and real literals to assign values to the data types real and double.

The following table describes each type of literal.

Type	Description

Date	The date, including the full year (1000 to 3000), the number of the month (01 to 12), and the day (01 to 31), separated by hyphens. For example:
	`1992-12-25   // December 25, 1992`
	`1995-02-06   // February 6, 1995`

Decimal	Any number with a decimal point and no exponent. The plus sign is optional (95 and +95 are the same). For numbers between zero and one, the zero to the left of the decimal point is optional (for example, 0.1 and .1 are the same). For whole numbers, zeros to the right of the decimal point are optional (32.00, 32.0, and 32. are all the same). For example:
	`12.34   0.005   14.0   15`
	`16.     -6500   +3.5555`

Integer	Any whole number (positive, negative, or zero). The leading plus sign is optional (18 and +18 are the same). For example: `1  123  1200  +55  -32`

Real	A decimal value, followed by E, followed by an integer; no spaces are allowed. The decimal number before the E follows all the conventions specified above for decimal literals. The leading plus sign in the exponent (the integer following the E) is optional (3E5 and 3E+5 are the same). For example:
	`2E4        2.5E78   +6.02E3   -4.1E-2`
	`-7.45E16   7.7E+8   3.2E-45`

String	As many as 1024 characters enclosed in single or double quotes, including a string of zero length or an empty string. For example: `"CAT"   "123"   'C:\WEST94'   ""`

Type	Description

Time	The time in 24-hour format, including the hour (00 to 23), minute (00 to 59), second (00 to 59), and fraction of second (up to six digits) with a range from 00:00:00 to 23:59:59.999999. You separate parts of the time with colons, except for fractional sections, which should be separated by a decimal point. For example:
	`21:09:15      // 15 seconds after 9:09 pm`
	`06:00:00      // Exactly 6 am`
	`10:29:59      // 1 second before 10:30 am`
	`10:29:59.9    // 1/10 sec before 10:30 am`

Using strings and chars

PowerBuilder provides two character-based data types: char and string. Chars contain one character; strings can contain multiple characters. You can define arrays of either type.

Strings

Most of the character-based data in your application, such as names, addresses, and so on, will be defined as strings. PowerScript provides many functions that you can use to manipulate strings, such as a function to convert characters in a string to uppercase and functions to remove leading and trailing blanks.

Chars

If you have character-based data that you will want to parse in an application, you might want to define it as an array of type char. Parsing a char array is easier and faster than parsing strings. Also, if you will be passing character-based data to external functions, you might want to use char arrays instead of strings.

☞ For more information about passing character-based data to external functions, see Chapter 6, "Functions."

Using quotation marks

You can use either single or double quotation marks with strings and chars. For example, these two assignments are equivalent.

```
string s1
s1 = "This is a string"
s1 = 'This is a string'
```

Similarly, these two assignments are equivalent.

```
char c
c = "T"
c = 'T'
```

You can embed a quotation mark in a string literal if you enclose the literal with the other quotation mark. For example:

```
string s1
s1 = "Here's a string."
```

results in the string *Here's a string.*

You can also use a tilde (~) to embed a quotation mark in a string literal. For example:

```
string s1 = 'He said, "It~'s good!"'
```

Complex nesting

When you nest a string within a string, which is nested in another string, you can use tildes to tell the parser how to interpret the quotation marks. Each pass through the parser strips away the outermost quotes and interprets the character after each tilde as a literal. Two tildes become one tilde and tilde-quote becomes the quote alone.

This string has two levels of nesting.

```
"He said ~"she said ~~~"Hi ~~~" ~" "
```

The first pass results in:

```
He said "she said ~"Hi ~" "
```

The second pass results in:

```
she said "Hi"
```

Finally, the third pass results in:

```
Hi
```

A more realistic example is a string for the Modify function that sets a DataWindow attribute. The argument string often requires complex quotation marks because you must specify one or more levels of nested strings. To figure out the quotation marks, consider how PowerBuilder will parse the string. The following string is a possible argument for the Modify function. It mixes single and double quotes to reduce the number of tildes.

```
"bitmap_1.Invert='0~tIf(empstatus=~~'A~~',0,1)'"
```

The double quotes tell PowerBuilder to interpret the argument as a string. It contains the expression being assigned to the Invert attribute, which is also a string, so it must be quoted. The expression itself includes a nested string, the quoted A. First, PowerBuilder evaluates the argument for Modify and assigns the single-quoted string to the Invert attribute. In this pass through the string, it converts two tildes to one. The string assigned to Invert becomes:

```
'0[tab]If(empstatus=~'A~',0,1)'
```

Finally, PowerBuilder evaluates the attribute's expression, converting tilde-quote to quote, and sets the bitmap's colors accordingly.

There are many ways to specify quotation marks for a particular set of nested strings. The following expressions for the Modify function all have the same end result.

```
"emp.Color = ~"0~tIf(stat=~~~"a~~~",255,16711680)~""
"emp.Color = ~"0~tIf(stat=~~'a~~',255,16711680)~""
"emp.Color = '0~tIf(stat=~~'a~~',255,16711680)'"
"emp.Color = ~"0~tIf(stat='a',255,16711680)~""
```

Rules for quotation marks and tildes

When nesting quoted strings, the following rules of thumb may help:

- A tilde tells the parser that the next character should be taken as a literal, not a string terminator.
- Pairs of single quotes (') can be used in place of pairs of tilde double quotes (~").
- Pairs of tilde tilde single quotes (~~') can be used in place of pairs of triple tilde double quotes (~~~").

Converting between strings and chars

There is no explicit char literal type. String literals convert to type char using the following rules:

- When a string literal is assigned to a char variable, the first character of the string literal is assigned to the variable. For example:

  ```
  char c = "xyz"
  ```

  results in the character *x* being assigned to the char variable c.

- Special characters (such as newline, formfeed, octal, hex, and so on) can be assigned to char variables using string conversion, such as:

  ```
  char c = "~n"
  ```

Also, string variables assigned to char variables convert using the same rules. A char variable assigned to a string variable results in a one-character string.

Assigning strings to char arrays

As with other data types, you can use arrays of chars. Assigning strings to char arrays follows these rules:

- If the char array is unbounded (that is, if it is defined as a variable-size array), the contents of the string are copied directly into the char array.
- If the char array is bounded and its length is less than or equal to the length of the string, the string is truncated in the array.
- If the char array is bounded and its length is greater than the length of the string, the entire string is copied into the array along with its zero terminator. Remaining characters in the array are undetermined.

Assigning char arrays to strings

When a char array is assigned to a string variable, the contents of the array are copied into the string up to a zero terminator, if found, in the char array.

Using both strings and chars in an expression

Expressions using both strings and char arrays promote the chars to strings before evaluation. For example:

```
char  c
.
.
.
if (c = "x") then
```

promotes the contents of c to a string before comparison with the string "x".

Using chars in PowerScript functions

All PowerScript functions that take strings also take chars and char arrays, subject to the conversion rules described above.

System object data types

In PowerBuilder applications, you manipulate objects such as windows, menus, command buttons, listboxes, and graphs. Internally, PowerBuilder defines each of these kinds of objects as a data type. Usually you don't need to concern yourself with these objects as data types — you simply define the objects in a PowerBuilder painter and use them.

But there are times when you need to understand how PowerBuilder maintains its system objects in a hierarchy of data types. For example, when you need to define instances of a window, you will define variables whose data type is window. When you need to create an instance of a menu to pop up in a window, you will define a variable whose data type is menu.

This section describes the PowerBuilder system object hierarchy.

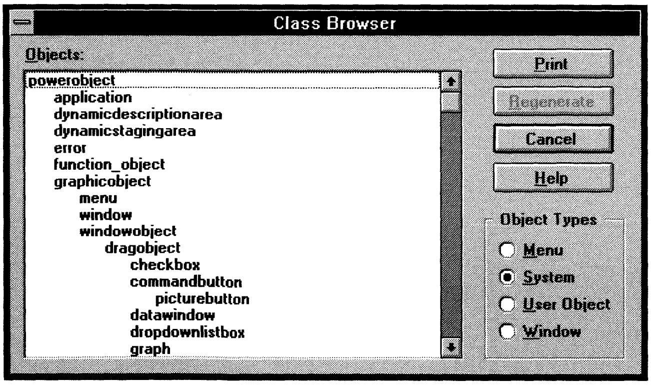
Using the Class browser

The easiest way to understand the hierarchy of system objects is to use the Class browser.

❖ To open the Class browser:

1. Open the Library painter.
2. Select Utilities➤Browse Class Hierarchy from the menu bar.

   The Class browser displays.



Objects:

```
powerobject
    application
    dynamicdescriptionarea
    dynamicstagingarea
    error
    function_object
    graphicobject
        menu
        window
        windowobject
            dragobject
                checkbox
                commandbutton
                    picturebutton
                datawindow
                dropdownlistbox
                graph
```

Object Types

○ Menu
● System
○ User Object
○ Window

3 Select the System button in the Object types box to see the system objects. (Clicking any of the other buttons displays the inheritance hierarchy of objects that have been created in the current application.)

About the system object hierarchy

PowerBuilder maintains its system objects in a class hierarchy. Each type of object is a class. The classes form an inheritance hierarchy of ancestors and descendants.

Looking at the hierarchy

By scrolling through the list of classes in the Class browser, you can see the hierarchy. The Class browser uses indentation to show inheritance. In the preceding screen, for example, you can see that at the top of the hierarchy is PowerObject—all PowerBuilder system objects are derived from PowerObject.

Looking further down the list, you see GraphicObject, which is the class that serves as the ancestor to all the graphical objects you use in PowerBuilder applications. For example, Menu is a type of GraphicObject—that is, the Menu class is derived from the GraphicObject class. Window is also a type of GraphicObject.

Objects as data types

All the classes shown in the Class browser are actually data types that you can use in your applications. You can define variables whose type is any class.

Examples

For example, to define a window variable, you could code:

```
window mywin
```

To define a menu variable, you could code:

```
menu mymenu
```

If you have a series of buttons in a window and for some reason need to keep track of one of them (for example, the last one clicked), you could declare a variable of type CommandButton and assign it the appropriate button in the window.

```
// Instance variable in a window
commandbutton LastClicked

// In Clicked event for a button in the window.
// Indicates that the button was the last one
// clicked by the user.
LastClicked = This
```

Because it is a CommandButton, the LastClicked variable has all the attributes of a CommandButton. After the last assignment above, LastClicked's attributes have the same values as the most recently clicked button in the window.

For more information

To learn more about working with instances of objects through data types, see the following chapters in the *User's Guide.*

Chapter	Describes
"Defining Windows"	Creating instances of windows
"Understanding Inheritance"	Using inheritance in an application
"Managing Libraries"	Using the Class browser

Enumerated data types

Like the system object data types, enumerated data types are specific to PowerScript. These data types are used in two ways:

- As arguments in functions
- To specify the attributes of an object or control

About enumerated data types

Each enumerated data type can be assigned a fixed set of values. Values of enumerated data types always end with an exclamation point (!).

For example, the enumerated data type Alignment, which specifies the alignment of text, can be assigned one of the following three values: Center!, Left!, and Right!.

When you enter enumerated data type values, do not enclose the value in quotation marks.

```
// This is correct.
mle_edit.Alignment = Left!

// The following statement will NOT compile.
// "Left!" is a string and the compiler
// expects an enumerated data value.
mle_edit.Alignment="Left!"
```

Advantage of enumerated types

Enumerated data types have the following advantage over standard data types: when an enumerated data type is required, the compiler checks the data and makes sure it is the correct type.

For example, to set the alignment of text in a line edit in a script, set the Alignment attribute to one of the Alignment enumerated data values, such as:

```
mle_edit.Alignment=Right!
```

If you set the Alignment attribute to any other data type or value, the compiler will not allow the value.

Listing the enumerated data types

You can list all the enumerated data types and their values in the Object browser.

❖ To list the enumerated data types:

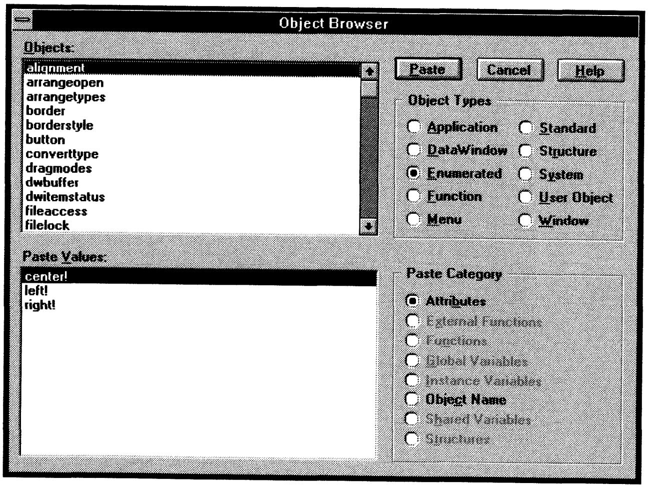
1 Do one of the following:

- ♦ Open the PowerScript painter and click the Browse icon or select Edit➤Browse Objects from the menu bar.
- ♦ Open the Library painter and select Utilities➤Browse Objects.

The Object browser opens.

2 Select Enumerated as the Object Type and Attributes as the Paste Category.

PowerBuilder lists all enumerated data types in the Objects box and the valid values of the selected data type in the Paste Values box.



Objects:

alignment
arrangeopen
arrangetypes
border
borderstyle
button
converttype
dragmodes
dwbuffer
dwitemstatus
fileaccess
filelock

Object Types

○ Application ○ Standard
○ DataWindow ○ Structure
● Enumerated ○ System
○ Function ○ User Object
○ Menu ○ Window

Paste Values:

center!
left!
right!

Paste Category

● Attributes
○ External Functions
○ Functions
○ Global Variables
○ Instance Variables
○ Object Name
○ Shared Variables
○ Structures

For more information

To learn how to	See
Use enumerated data types in attribute assignments	*Objects and Controls*, which lists all attributes of the PowerBuilder objects and controls
Use enumerated data types in PowerScript functions	*Function Reference*
Use the Object browser	Chapter 3, "Writing Scripts," in the *User's Guide*

CHAPTER 3

Declarations

About this chapter

Before you use a variable or array in a script, you must declare it (give it a type and a name). For example, before you can use an integer variable, you must identify it as an integer and assign it a name.

This chapter explains how to declare variables and arrays.

Contents

Types of variables

PowerScript recognizes four types of variables:

- Global variables, which are accessible anywhere in an application
- Instance variables, which are associated with one instance of an object, such as a window
- Shared variables, which are associated with a type of object
- Local variables, which are accessible only in one script

Global variables

You use global variables when you have data that needs to be available anywhere: global variables can be used without qualification in any script in an application.

For example, if you have defined a global integer variable named WinCount, you can reference the variable directly in any script, such as:

```
WinCount = WinCount + 1
```

❖ To declare global variables:

- Select Declare➤Global Variables in the Window, User Object, Menu, or PowerScript painter.

Instance variables

You use instance variables when you have variables that need to be accessible in more than one script within an object, but that don't need to be global throughout the entire application. For example, several scripts for a window might reference an employee ID. You can declare EmpID as an instance variable for that window; all scripts in that window have access to that variable. In effect, instance variables are attributes of the object.

Instance variables can be application-level, window-level, user-object-level, or menu-level variables:

- Application-level variables are declared within the application object.

  They are always available in any scripts for the application object. In addition, you can make them public so that they are accessible throughout the application.

- Window-level variables are declared within a window.

  They are always available in any scripts for the window in which they are declared and the controls in that window. In addition, you can make them public so that they are accessible throughout the application.

- User-object-level variables are declared within a user object.

  They are always available in any scripts for the user object in which they are declared and the controls in that user object. In addition, you can make them public so that they are accessible throughout the application.

- Menu-level variables are declared within a menu.

  They are always available in any scripts for the menu in which they are declared and its MenuItems. In addition, you can choose to make access to them available throughout the application.

Declaring instance variables

❖ To declare instance variables:

- Select Declare➤Instance Variables in the Window, User Object, Menu, or PowerScript painter.

Specifying access to instance variables

When you declare an instance variable you can also specify the **access level** for the variable—that is, you can specify which scripts have access to the instance variable.

Access	You can reference the instance variable in
Public	Any script in the application.
Private	Scripts for events in the object for which the variable is declared. You cannot reference the variable in descendants of the object.
Protected	Scripts for the object for which the variable is declared and its descendants.

To specify an access level when you declare an instance variable, include the access level in the declaration. If you don't specify an access level, the variable is defined as Public.

Two ways to specify access

You can specify the access level using one of two formats. In the first format, you include the access specifier on the same line as the declaration, before the data type.

```
access-specifier type variablename
access-specifier type variablename
. . .
```

For example:

```
private integer  a, n
public integer  Subtotal
protected integer  WinCount
```

In the second format, you can group declarations by including the access specifier on its own line, followed by a colon (:).

```
access-specifier:
   type variablename
   type variablename
```

For example:

```
Private:
   integer   a=10, b=24
   string  Name, Address1
Protected:
   integer Units
   double  Results
   string  Lname
```

```
Public:
   integer Weight
   string  Location="Home"
```

In the preceding example, a, b, Name, and Address1 are Private variables; Units, Results, and Lname are Protected variables; and Weight and Location are Public variables.

For more information about declaring variables of different data types, see "Declaring variables" on page 41.

How instance variables are initialized

When you define an instance variable for a window, menu, or application object, the instance variable is initialized when the object is opened. Its initial value is the default value for its data type or the value specified in the variable declarations.

When you close the object, the instance variable ceases to exist. If you open the object again, the instance variable is initialized again.

Tip
If you need a variable that continues to exist after the object is closed, use a shared variable (see "Shared variables" on page 39).

When using multiple instances of windows

When you build a script for one of multiple instances of a window, instance variables can have a different value in each instance of the window. For example, to set a flag based on the contents of the instance of a window, you would use an instance variable.

Tip
If you need a variable that keeps the same value over multiple instances of an object, use shared variables (see "Shared variables" on page 39).

Referring to instance variables

You can refer to instance variables in scripts if there is an instance of the object open in the application. Depending on the situation, you might need to qualify the name of the instance variable with the name of the object defining it.

Using unqualified names

You can refer to instance variables without qualifying them with the object name in the following cases:

- For application-level variables, in scripts for the application object
- For window-level variables, in scripts for the window itself and in scripts for controls in that window
- For user-object-level variables, in scripts for the user object itself and in scripts for controls in that user object
- For menu-level variables, in scripts for the menu itself and in scripts for the MenuItems in that menu

For example, if w_emp has an instance variable EmpID, in any script for w_emp or its controls, you can reference EmpID without qualification, such as:

```
sle_id.Text = EmpID
```

Using qualified names

In all other cases, you need to qualify the name of the instance variable with the name of the object using dot notation as follows.

object.instance-variable

(Of course, this applies only to Public or Protected instance variables. You cannot reference Private instance variables outside the object at all.)

For example, to refer to the w_emp instance variable EmpID from a script outside the window, you need to qualify the variable with the window name, such as:

```
sle_ID.Text = w_emp.EmpID
```

There is another situation in which references must be qualified: suppose that w_emp has an instance variable EmpID and that in w_emp there is a command button that declares a *local* variable EmpID in its Clicked script. In that script, you must qualify all references to the instance variable, such as:

```
Parent.EmpID
```

Shared variables

Shared variables, like instance variables, can be application-level, window-level, user-object level, or menu-level variables. Shared variables are associated with the object definition, rather than an instance of the object. Therefore, all instances of the object type have the shared variable in common.

For example, if you define a shared variable for the window w_emp, each instance of w_emp open in the application uses the same variable: the value of the shared variable is the same in each instance of w_emp.

Shared variables retain their value when an object is closed and then opened again.

Shared variables are always private. You can access a shared variable only in scripts for the object for which the variable is declared, including scripts for controls associated with the object. You cannot reference the variable in descendants of the object. If you require more general access to the variable, you can make it global instead.

❖ To declare shared variables:

♦ Select Declare➤Shared Variables in the Window, User Object, Menu, or PowerScript painter.

Declaring a shared variable is similar to declaring an instance variable, except there is no access specifier. You specify only the type and the variable name.

type variablename

For example:

```
integer  Subtotal
integer  WinCount
```

You reference shared variables the same way you reference instance variables (see page 37).

For more information about declaring variables of different data types, see "Declaring variables" on page 41.

How shared variables are initialized

When you use a shared variable in the script for a window or menu, the variable is initialized when the first instance of the window is opened. When you close the window, the shared variable continues to exist until you exit the application. If you open the window again without exiting the application, the shared variable will have the value it had when you closed the window.

For example, if in the script for a window you set the shared variable Count to 20 and close the window, and then reopen the window without exiting the application, Count will be equal to 20.

When using multiple instances of windows
If you have multiple instances of the window in the example above, Count will be equal to 20 in each instance. Since shared variables are shared among all instances of the window, changing Count in any instance of the window changes it for all instances.

Local variables

Use local variables when you need a temporary variable to hold some value. Local variables are declared in a script and can be used only in that script.

How PowerBuilder looks for variables

When PowerBuilder executes a script and finds an unqualified reference to a variable, it searches for the variable in the following order:

1. A local variable
2. A shared variable
3. A global variable
4. An instance variable

As soon as PowerBuilder finds a variable with the specified name, it uses the variable's value.

Declaring variables

There are two sets of syntax for declaring variables: a standard syntax for all variable data types except blob and decimal, and a syntax for blob and decimal variables.

Standard declarations

To declare any variable except a blob or decimal, enter the data type followed by one or more spaces and the variable name:

type variablename

Examples

```
int count                   // Declares count as an
                            // as a long

string first-name           // Declares first-name as
                            // a string
                            // Strings do not have
                            // predefined sizes
```

You can declare multiple variables of the same data type on one line. To declare additional variables of the same type on the same line, enter a comma and the next variable name.

```
int a, b, c                 // Declares a, b, and c
```

X and Y as variable names
Although you may think of x and y as typical variable names, in PowerBuilder they are also attributes that specify an object's onscreen coordinates. If you use them as variables and forget to declare them, you will not get a compiler error. PowerBuilder will assume you want to move the object, which may lead to interesting activity in your application.

Blob declarations

To declare a blob variable, enter **Blob** followed by the length of the blob (in bytes) enclosed in braces ({ }) and the variable name. The length is optional, and braces are required only if you specify the length.

blob {*size*} *variablename*

If you enter the length and exceed the declared length in a script, PowerBuilder will truncate the blob. If you do not enter the length in the declaration, the blob has an initial length of 0 and PowerBuilder will adjust its size each time it is used at execution time.

Blobs cannot be initialized with a value. Only their size can be initialized.

Examples

```
blob        Emp_Picture    // Declares Emp_Picture
                           // a blob with 0 length

blob{100}   Emp_Picture    // Declares Emp_Picture
                           // a blob with a length of
                           // 100 bytes
```

Decimal declarations

To declare a decimal variable, enter **Dec** or **Decimal** followed by the number of digits after the decimal point (the **precision**) enclosed in braces ({ }) and the variable name. The braces are required only if you enter the precision.

decimal {*precision*} *variablename*

If you do not enter the precision in the declaration, the variable takes the precisions assigned to it in the script.

Examples

```
decimal{2} Amount     // Declares Amount as a
                      // decimal number with 2
                      // digits after the
                      // decimal point

dec{4} Rate1, Rate2   // Declares Rate1 and
                      // Rate2 as decimal
                      // numbers with 4
                      // digits after the
                      // decimal point

decimal{0} Balance    // Declares Balance as a
                      // decimal with 0 digits
                      // after the decimal point

dec Result
dec{2} Op1, Op2
Result = Op1 * Op2    // Result now has 4 digits
                      // after the decimal point
```

Initial values

When you declare a variable, you can assign an initial value to the variable or accept the default initial value.

Assigning values

To assign a value to a variable when you declare it, place an equal sign (=) and a literal appropriate for that variable data type after the variable.

Examples

```
int    count=5    // Declares count as an integer
                  // and assigns 5 to it

int    a=5, b=10   // Declares a and b as integers
                   // and assigns 5 to a and 10 to b

string   method="UPS" // Declares method as a
                      // string and assigns
                      // "UPS" to it

int    a=1, b, c=100   // Declares a, b, and c
                       // as integers, assigns 1 to
                       // a, lets b default to 0,
                       // and assigns 100 to c

date StartDate = 1993-02-01  // Declares StartDate
                             // as a date and
                             // assigns Feb 1, 1993,
                             // to it
```

Initializing a variable with an expression

You can initialize a variable with the value of an existing variable or expression, such as:

```
integer i = 100
integer j = i
```

When you do this, the second variable is assigned the value of the expression when the script is compiled. The initialization is not reevaluated during execution.

This is an important point if the value of the expression will change based on current conditions. For such values, declare the variable and assign the value in separate statements.

For example, in the following declaration, the value assigned to d is the date the script is compiled, not the date the application is run.

```
date d = Today( )
```

In contrast, the following statements result in d being set to the date the application is run.

```
date d
d = Today( )
```

Using default values

If you do not assign a value to a variable when you declare it, PowerBuilder sets the variable to the default value for its data type.

The following table lists the default values for variable data types.

Variable data type	Default value
Blob	A blob of 0 length; an empty blob
Char	ASCII value 0
Boolean	FALSE
Date	1900-01-01 (January 1, 1900)
DateTime	1900-01-01 00:00:00
Numeric (integer, long, decimal, real, double, UnsignedInteger, and UnsignedLong)	0
String	Empty string ("")
Time	00:00:00 (midnight)

Declaring arrays

An array is an indexed collection of elements of a single data type. An array can be single- or multidimensional. Single-dimensional arrays can have a fixed or variable size, and single-dimensional arrays without a range can have approximately two gigabytes of elements. Each dimension of a multidimensional array can have two gigabytes of elements.

To declare an array, include square brackets after the variable name. To declare a fixed-size array, include the sizes of the array in the square brackets. For a multidimensional array, there will be a size for each dimension.

Fixed-size arrays

When you declare a fixed-size array you specify its size. You can specify how the elements in the array are numbered with the TO notation and you can initialize the array elements with defaults values.

Here is an example of a single-dimensional array of three integers named TaxCode:

```
int TaxCode[3]  // Declares an array of 3 integers
```

To refer to individual array elements, use square brackets and the element number, such as TaxCode[1], TaxCode[2], and TaxCode[3].

Default values for array elements

PowerBuilder initializes each element of an array to the same default value as its underlying data type. For example, in the integer array TaxCode[3], the elements TaxCode[1], TaxCode[2], and TaxCode[3] are all initialized to zero.

To override the default values, initialize the elements of the array when you declare the array by specifying a comma-separated list of values enclosed in braces. Here is an example of an initialized one-dimensional array of three variables:

```
real Rate[3]={1.20, 2.40, 4.80}
```

Tip

You can assign values after declaring an array using the same syntax.

```
integer Arr[ ]
Arr = {1, 2, 3, 4}
```

Array element numbering

Array elements start counting at 1 (TaxCode[1]). To override this default, use the TO notation. The TO notation only applies to fixed-size arrays.

```
real Rate[2 to 5]     // Declares array of 4 real
                      // numbers:  Rate[2], Rate[3],
                      // Rate[4] and Rate[5]

int Qty[0 to 2]       // Declares array of 3 integers

string Test[-2 to 2] // Declares 5 strings
```

In an array dimension, the second number must be greater than the first. These declarations are invalid.

```
int   count[10 to 5]       // INVALID because 10 is
                           // greater than 5

int   price[-10 to -20]    // INVALID because -10 is
                           // greater than -20
```

Variable-size arrays

A variable-size array consists of a variable name followed by square brackets but no number. PowerBuilder defines it *by use* at execution time (subject only to memory constraints). Only one-dimensional arrays can be variable-size arrays.

Because you don't declare the size, you can't use the TO notation to change the lower bound of the array. Therefore, the lower bound of a variable-size array is always 1.

Examples

This example declares a variable-size array and assigns values to three array elements.

```
long   price[ ]  // Declares a variable-size
                 // array of any quantity of
                 // decimal numbers
```

```
price[100]=2000
price[50] =3000
price[110]=5000
```

When the statements above first execute, they allocate memory as follows:

- The statement price[100]=2000 will allocate memory for 100 long numbers price[1] to price[100], then assign 0 (the default for numbers) to price[1] through price[99] and assign 2000 to price[100].
- The statement price[50]=3000 will not allocate more memory, but will assign the value 3000 to the 50th element of the price array.
- The statement price[110]=5000 will allocate memory for 10 more long numbers named price[101] to price[110], then assign 0 (the default for numbers) to price[101] through price[109] and assign 5000 to price[110].

To initialize a variable-size array, list all required values in braces. The following statement sets code[1] equal to 11, code[2] equal to 242, and code[3] equal to 27.

```
int code[ ]={11,242,27}
```

Multidimensional arrays

A fixed-size array can have more than one dimension. To specify additional dimensions, use a comma-separated list. The amount of memory in your system is the only limit to the number of dimensions for an array. You cannot initialize multidimensional arrays.

Example

Here is an example of a declared six-element two-dimensional integer array.

```
int score[2,3]  // Declares a 6-element,
                // 2-dimensional array
```

The individual elements are score[1,1], score[1,2], score[1,3], score[2,1], score[2,2], and score[2,3].

Index values

By default, all index values of a multidimensional array start at 1, but you can override the default with the TO notation.

Examples

The array declarations below are valid.

```
// 2-dimensional 75-element array
int    RunRate[1 to 5, 10 to 25]

// 3-dimensional 45,000-element array
long   days[3,300,50]

// 3-dimensional 20,000-element array
int    staff[100,0 to 20,-5 to 5]
```

String arrays

You declare string arrays the same way you declare numeric arrays.

Examples

```
string   day[7]              // Declares a one-
                             // dimensional array
                             // of 7 strings

string   name[-10 to 15]     // Declares a one-
                             // dimensional array
                             // of 26 strings

string   plant[3,10]         // Declares a 2-
                             // dimensional array
                             // of 30 strings

string   city[ ]             // Declares an array that
                             // can hold any number of
                             // strings and each string
                             // can be any length
```

Decimal arrays

To declare a decimal array, enter **Dec** or **Decimal**, followed by the number of digits after the decimal point (the **precision**) enclosed in braces ({ }), the array name, and the dimensions of the array enclosed in square brackets.

If you do not enter the precision in the declaration, the variable takes the precisions assigned to it in the script.

Examples

```
dec{2}   Cost[10]     // Declares an array of
                      // 10 decimal numbers
                      // each with 2 digits
                      // following the decimal
                      // point

decimal   price[20] // Declares an array of
                      // 20 decimal numbers where
                      // each takes the assigned
                      // precision; no precision
                      // specified

dec{8}   limit[ ]     // Declares a variable-size
                      // array of decimal numbers
                      // each with 8 digits
                      // following the decimal
                      // point

dec   limit[ ]        // Declares a variable-size
                      // array of decimal numbers
                      // and does not specify the
                      // precision so each element
                      // will take the precision
                      // assigned

dec{2}   rate[3,4]    // Declares a 2-dimensional
                      // array of 12 decimal numbers
                      // each with 2 digits after
                      // the decimal point

decimal{3}   first[10],second[15,5],third[ ]

                      // The line above declares
                      // 3 decimal arrays.
                      // Every number in each
                      // array has 3 digits
                      // after the decimal point.
```

Array errors

Referring to array elements outside the declared size causes an error during execution. For example:

```
int   test[10]
test[11]=50      // This causes an execution error.
test[0]=50       // This causes an execution error.

int   trial[5,10]
trial [6,2]=75   // This causes an execution error.
trial [4,11]=75  // This causes an execution error.
```

Accessing a variable-size array above its largest assigned value or below its lowest assigned value also causes an error during execution.

```
int   stock[ ]
stock[50]=200

if stock[51]=0 then Beep(1)  // This causes
                             // an execution error.
if stock[0]=0 then Beep(1)   // This causes
                             // an execution error.
```

CHAPTER 4
Operators and Expressions

About this chapter

Operators perform arithmetic calculations; compare numbers, text, and boolean values; execute logical operations on boolean values; and concatenate strings and blobs.

This chapter describes the operators supported in PowerScript and how to use them in expressions.

Contents

Operators

PowerScript supports the following types of operators:

- Arithmetic
- Relational
- Logical
- Concatenation

Arithmetic operators

The following table lists the arithmetic operators.

Operator	Meaning	Example
+	Addition	Total=SubTotal+Tax
-	Subtraction	Price=Price - Discount Unless you have prohibited the use of dashes in identifier names, you must surround the minus sign with spaces. For more information, see "Identifier names" in Chapter 1, "Language Basics."
*	Multiplication	Total=Quantity*Price
/	Division	Factor=Discount/Price
^	Exponentiation	Rank=Rating^2.5

Multiplication and division

Multiplication and division are carried out to full precision (16–18 digits). Decimal numbers are rounded (not truncated) on assignment.

Examples

These examples show the values that result from various operations on decimal values.

```
decimal {4} a,b,d,e,f
decimal {3} c

a = 20.0/3                  // a contains  6.6667
b = 3 * a                   // b contains 20.0001
```

```
c = 3 * a                 // c contains 20.000
d = 3 * (20.0/3)          // d contains 20.0000
e = Truncate(20.0/3, 4) // e contains  6.6666
e = Truncate(20.0/3, 5) // e contains  6.6667
```

Subtraction

If the PowerBuilder preferences variable DashesInIdentifiers is set to 1, then you must always surround the subtraction operator and the -- operator with spaces. Otherwise, PowerBuilder interprets the expression as an identifier.

For example:

```
A - B     // Always means subtract B from A

A-B       // Means a variable named A-B
          // if DashesInIdentifiers is set to 1
          // but means subtract B from A if
          // DashesInIdentifiers is set to 0
```

☞ For information about setting DashesInIdentifiers, see "Identifier Names" in Chapter 1, "Language Basics." For information about the -- operator, see "Assignment statements" in Chapter 5, "Statements."

Calculations with NULL

When you form an arithmetic expression that contains a NULL value, the expression becomes NULL.

> **Tip**
> Thinking of NULL as undefined makes this easier to understand.

Examples

When the value of variable c is NULL, the following assignment statements all set the variable a to NULL.

```
integer a, b=100, c

SetNull(c)

a = b+c       // a is NULL
a = b - c     // a is NULL
a = b*c       // a is NULL
a = b/c       // a is NULL
```

☞ For more information about NULL values, see Chapter 1, "Language Basics."

Errors and overflows

Division by zero, exponentiation of negative values, and so on, cause errors during execution.

Overflow of real, double, and decimal values cause errors during execution. Overflow of signed or unsigned integers and longs cause results to wrap.

Example

This example illustrates how the value of the variable i after overflow occurs.

```
integer i
i = 32767
i = i + 1      // i is now -32768
```

Relational operators

PowerBuilder uses relational operators in relational expressions to evaluate two or more operands. The result is always TRUE or FALSE.

The following table lists the relational operators.

Operator	Meaning	Example
=	Equals	if Price=100 then Rate=.05
>	Greater than	if Price>100 then Rate=.05
<	Less than	if Price<100 then Rate=.05
<>	Not equal	if Price<>100 then Rate=.05
>=	Greater than or equal	if Price>=100 then Rate=.05
<=	Less than or equal	if Price<=100 then Rate=.05

Comparing strings

When PowerBuilder compares strings, the comparison is case-sensitive. Trailing blanks are significant.

Case-sensitive examples

If you compare two strings with the same text but different case, the comparison fails. But if you use the Upper or Lower function, you can ensure that the case of both strings are the same so that only the content affects the comparison:

```
City1="Austin"
City2="AUSTIN"
if City1=City2 ...                  // Will return FALSE
```

```
City1="Austin"
City2="AUSTIN"
if Upper(City1)=Upper(City2)... // Will return TRUE
```

Tip

To compare strings regardless of case, use the Upper or Lower function. For information about these functions, see the *Function Reference.*

Trailing blanks examples

In this example, trailing blanks in one string cause the comparison to fail:

```
City1="Austin"
City2="Austin        "
if City1=City2 ...               // Will return FALSE
```

Tip

To remove trailing blanks, use the RightTrim function. To remove leading blanks, use the LeftTrim function. To remove leading and trailing blanks, use the Trim function. For information about these functions, see the *Function Reference.*

Logical operators

PowerBuilder uses logical operators to form boolean expressions. The result of evaluating a boolean expression is always TRUE or FALSE.

The following table lists the logical operators.

Operator	Meaning	Example
NOT	Logical negation	if NOT Price=100 then Rate=.05
AND	Logical and	if Tax>3 AND Ship<5 then Rate=.05
OR	Logical or	if Tax>3 OR Ship<5 then Rate=.05

NULL value evaluations

When you form a boolean expression that contains a NULL value, the AND and OR operators behave differently. Thinking of NULL as undefined (neither TRUE nor FALSE) makes the results easier to calculate.

Examples

```
boolean d, e=TRUE,f
SetNull(f)
d=e and f    // d is NULL
d=e or f     // d is TRUE
```

☞ For more information about NULL values, see Chapter 1, "Language Basics."

Concatenation operator

The concatenation operator joins the contents of two variables of the same type to form a longer value. You can concatenate strings and blobs.

To concatenate values, use the plus sign (+) operator.

Examples

These examples concatenate several strings.

```
string Test
Test = "over" + "stock" // Test contains "overstock"

string Lname, Fname, FullName
FullName = Lname + ', ' + Fname
      // FullName contains last name and first name,
      // separated by a comma and space.
```

This example shows how a blob can act as an accumulator when reading data from a file.

```
integer i, fnum, loops
blob tot_b, b
. . .
FOR i = 1 to loops
   bytes_read = FileRead(fnum, b)
   tot_b = tot_b + b
NEXT
```

Operator precedence in expressions

To ensure predictable results, all operators in a PowerBuilder expression are evaluated in a specific order of precedence. When the operators have the same precedence, PowerBuilder evaluates them left to right.

The following table lists the operators in descending order of precedence.

Operator	Purpose
( )	Grouping (see note below)
+, -	Unary plus and unary minus
^	Exponentiation
*, /	Multiplication and division
+, -	Addition and subtraction; string concatenation
=, >, <, <=, >=, <>	Relational operators
NOT	Negation
AND	Logical and
OR	Logical or

Grouping expressions
To override the order, enclose expressions in parentheses. This identifies the group and order in which PowerBuilder will evaluate the expressions. When there are nested groups, the groups are evaluated from the inside out.

Example

In the expression (x+(y*(a+b))), a+b is evaluated first. The sum of a and b is then multiplied by y, and this product is added to x.

CHAPTER 5

Statements

About this chapter

This chapter describes the statements in PowerScript and how to use them in scripts.

Contents

Assignment statements

Use assignment statements to assign values to variables. To assign a value to a variable anywhere in a script, use the equal sign (=). For example:

```
String1 = "Part is out of stock"
TaxRate = .05
```

No multiple assignments

Since the equal sign is also a logical operator, you cannot assign more than one variable in a single statement. For example, the following statement does *not* assign the value 0 to A and B.

```
A=B=0     // This will not assign 0 to A and B.
```

The above statement first evaluates B=0 to TRUE or FALSE and then tries to assign this boolean value to A. When A is not a boolean variable, this line produces an error when compiled.

Assigning array values

You can assign multiple array values with one statement, such as:

```
int Arr[]
Arr = {1, 2, 3, 4}
```

You can also copy array contents. For example:

```
Arr1 = Arr2
```

copies the contents of Arr2 into array Arr1.

Shortcuts

PowerScript provides the following shortcuts you can use to assign values to variables. They have slight performance advantages over their equivalents.

Assignment	Example	Equivalent to
++	i ++	i = i + 1
--	i --	i = i - 1
+=	i += 3	i = i + 3
-=	i -= 3	i = i -3
*=	i *= 3	i = i * 3
/=	i /= 3	i = i / 3
^=	i ^=3	i = i ^ 3

Unless you have prohibited the use of dashes in variable names, you must leave a space before -- and -= (otherwise, PowerScript thinks the minus sign is part of a variable name).

For more information, see "Identifier names" in Chapter 1, "Language Basics."

Examples

Here are some examples of assignments.

```
int i = 4
i ++          // i is now 5.
i --          // i is 4 again.
i += 10       // i is now 14.
i /= 2        // i is now 7.
```

These shortcuts can be used only in pure assignment statements. They cannot be used with other operators in a statement. For example, the following is invalid.

```
int i, j
i = 12
j = i ++      // INVALID
```

The following is valid, because ++ is used by itself in the assignment.

```
int i, j
i = 12
i ++
j = i
```

Using dot notation

To assign a value to an attribute of an object, use PowerScript dot notation to identify the object and attribute.

object.attribute

where *object* is the name of the object (or the reserved word Parent, ParentWindow, or This), and *attribute* is the attribute to which you assign the value. You also use dot notation to test for or obtain the value of an object.

Examples

This example makes a CheckBox invisible.

```
Chkbox_on.Visible=FALSE
```

This example tests the value of the string in the SingleLineEdit sle_emp.

```
If sle_emp.Text="N" then Open(win_1)
```

This example calculates the value for the string Text1.

```
string Text1
Text1=sle_emp.Text+".DAT"
```

CALL

Description

CALL calls an ancestor script from a script for a descendant object. You can call scripts for events in an ancestor of the user object, menu, or window. You can also call scripts for events for controls in an ancestor of the user object or window.

Syntax

CALL *ancestorobject* {`*controlname*}::*event*

Parameter	Description
ancestorobject	An ancestor of the descendant object
controlname	The name of a control in an ancestor window or custom user object
event	An event in the ancestor object

Examples

The following statement calls a script for an event in an ancestor window.

```
CALL w_emp::Open
```

The following statement calls a script for an event in a control in an ancestor window.

```
CALL w_emp`cb_close::Clicked
```

👓 In some circumstances, you can use the Super reserved word when *ancestorobject* is the descendant object's immediate ancestor. See the discussion of Super in Chapter 1, "Language Basics."

CHOOSE CASE

Description

The CHOOSE CASE control structure directs program execution based on the value of a test expression (usually a variable).

Syntax

```
CHOOSE CASE testexpression
CASE expressionlist
    statementblock
{CASE expressionlist
    statementblock
. . .
CASE expressionlist
    statementblock}
{CASE ELSE
    statementblock}
END CHOOSE
```

Parameter	Description
testexpression	The expression on which you want to base the execution of the script
expressionlist	One of the following expressions: ♦ A single value ♦ A list of values separated by commas (for example, 2, 4, 6, 8) ♦ A TO clause (for example, 1 TO 30) ♦ IS followed by a relational operator and comparison value (for example, IS>5) ♦ Any combination of the above with an implied OR between expressions (for example, 1, 3, 5, 7, 9, 27 TO 33, IS >42)
statementblock	The block of statements you want PowerBuilder to execute if the test expression matches the value in *expressionlist*

Usage

At least one CASE clause is required. You must end a CHOOSE CASE control structure with END CHOOSE.

If *testexpression* at the beginning of the CHOOSE CASE statement matches a value in *expressionlist* for a CASE clause, the statements immediately following the CASE clause are executed. Control then passes to the first statement after the END CHOOSE clause.

If multiple CASE expressions exist, then *testexpression* is compared to each *expressionlist* until a match is found or the CASE ELSE or END CHOOSE is encountered.

If there is a CASE ELSE clause and the test value does not match any of the expressions, *statementblock* in the CASE ELSE clause is executed. If no CASE ELSE clause exists and a match is not found, the first statement after the END CHOOSE clause is executed.

Examples

This example provides different processing based on the value of the variable Weight.

```
CHOOSE CASE Weight

CASE IS<16
   Postage=Weight*0.30
   Method="USPS"

CASE 16 to 48
   Postage=4.50
   Method="UPS"

CASE ELSE
   Postage=25.00
   Method="FedEx"

END CHOOSE
```

This example converts the text in a SingleLineEdit control to a real value and provides different processing based on its value.

```
CHOOSE CASE Real(sle_real.Text)
CASE is < 10.99999
   sle_message.Text = "Real Case < 10.99999"

CASE 11.00 to 48.99999
   sle_message.Text = "Real Case 11 to 48.9999

CASE is > 48.9999
   sle_message.Text = "Real Case > 48.9999"

CASE ELSE
   sle_message.Text = "Cannot evaluate!"

END CHOOSE
```

CONTINUE

Use the CONTINUE statement in a DO...LOOP or a FOR...NEXT control structure. CONTINUE takes no parameters.

In a DO...LOOP structure

When PowerBuilder encounters a CONTINUE statement in a DO...LOOP, control passes to the next LOOP statement. The statements between the CONTINUE statement and the LOOP statement are skipped in the current iteration of DO...LOOP. In a nested DO...LOOP structure, a CONTINUE statement bypasses statements in the *current* DO...LOOP structure.

Example

The following statements display a message box twice: when B equals 2 and when B equals 3. As soon as B is greater than 3, the statement following CONTINUE is skipped during each iteration of the loop.

```
int A=1, B=1
DO WHILE A < 100
   A = A+1
   B = B+1
   if B > 3 then CONTINUE
   MessageBox("Hi", "B is " + String(B) )
LOOP
```

In a FOR...NEXT structure

When PowerBuilder encounters a CONTINUE statement in a FOR...NEXT control structure, control passes to the following NEXT statement; the statements between the CONTINUE statement and the NEXT statement are skipped in the current iteration of FOR...NEXT.

Example

The following statements stop incrementing B as soon as Count is greater than 15.

```
int A=0, B=0, Count
FOR Count = 1 to 100
   A = A + 1
   IF Count > 15 then CONTINUE
   B = B + 1
NEXT
// Upon completion, a=100 and b=15.
```

CREATE

Description

The CREATE statement generates an object instance for a specified object type. After a CREATE statement, attributes of the created object instance can be referenced using dot notation.

The CREATE statement returns an object instance which can be stored in a variable of the same type.

Syntax

objectvariable = CREATE *objecttype*

Parameter	Description
objectvariable	A global, instance, or local variable whose data type is *objecttype*
objecttype	The object data type

Usage

Use CREATE as the first reference to any Class user object. This includes standard Class user objects, such as mailSession or Transaction.

The system provides one instance of several standard Class user objects: Message, Error, Transaction, DynamicDescriptionArea, and DynamicStagingArea. You only need to use CREATE if you declare additional instances of these objects.

If you need a menu that is not part of an open window definition, use CREATE to create an instance of the menu. (See the PopMenu function in *Function Reference.*)

Use the appropriate Open function, instead of CREATE, to create an instance of a visual user object or window.

You do not need to use CREATE to allocate memory for a standard data type, such as integer or string, or any object that is not a class, such as the Environment object. You can use the Class browser to find out if an object you see in the Object browser is also a class.

Example

This example creates a new transaction object and stores the object in the variable DBTrans.

```
transaction DBTrans
DBTrans = CREATE transaction
DBTrans.DBMS = 'ODBC'
```

DESTROY

Description

DESTROY eliminates an object instance that was created with the CREATE statement. After a DESTROY statement, attributes of the deleted object instance can no longer be referenced.

Syntax

DESTROY *objectvariable*

Parameter	Description
objectvariable	A variable whose data type is a PowerBuilder object

Example

The following statement destroys the transaction object DBTrans that was created with a CREATE statement.

```
DESTROY DBTrans
```

DO...LOOP

The DO...LOOP control structure is a general-purpose iteration statement. Use DO...LOOP to execute a block of statements while or until a condition is true. DO... LOOP has four formats.

In all four formats of the DO...LOOP control structure, DO marks the beginning of the statement block that you want to repeat. The LOOP statement marks the end.

You can nest DO...LOOP control structures.

Using as DO UNTIL

Description

DO UNTIL...LOOP executes a block of statements until the specified condition is TRUE. If the condition is TRUE on the first evaluation, the statement block does not execute.

Syntax

```
DO UNTIL condition
    statementblock
LOOP
```

Parameter	Description
condition	The condition you are testing
statementblock	The block of statements you want to repeat

Using as DO WHILE

Description

DO WHILE...LOOP executes a block of statements while the specified condition is TRUE. The loop ends when the condition becomes FALSE. If the condition is FALSE on the first evaluation, the statement block does not execute.

Syntax

```
DO WHILE condition
    statementblock
LOOP
```

Parameter	Description
condition	The condition you are testing
statementblock	The block of statements you want to repeat

Using as LOOP UNTIL

Description

LOOP...UNTIL executes a block of statements at least once and continues until the specified condition is TRUE.

Syntax

```
DO
    statementblock
LOOP UNTIL condition
```

Parameter	Description
statementblock	The block of statements you want to repeat
condition	The condition you are testing

Using as LOOP WHILE

Description

LOOP...WHILE executes a block of statements at least once and continues while the specified condition is TRUE. The loop ends when the condition becomes FALSE.

Syntax

```
DO
   statementblock
LOOP WHILE condition
```

Parameter	Description
statementblock	The block of statements you want to repeat
condition	The condition you are testing

When to use the different forms

Use DO WHILE or DO UNTIL when you want to execute a block of statements *only* if a condition is TRUE (for WHILE) or FALSE (for UNTIL). DO WHILE and DO UNTIL test the condition *before* executing the block of statements.

Use LOOP WHILE or LOOP UNTIL when you want to execute a block of statements *at least once*. LOOP WHILE and LOOP UNTIL test the condition *after* the block of statements has been executed.

Examples

The following DO UNTIL executes a block of Beep functions until A is greater than 15.

```
integer A = 1, B = 1

DO UNTIL A > 15
   Beep(A)
   A = (A + 1) * B
LOOP
```

The following DO WHILE executes a block of BEEP functions only while A is less than or equal to 15.

```
integer A = 1, B = 1

DO WHILE A <= 15
   Beep(A)
   A = (A + 1) * B
LOOP
```

The following LOOP UNTIL executes a block of Beep functions and then continues to execute the functions until A is greater than 15.

```
integer A = 1, B = 1

DO
   Beep(A)
   A = (A + 1) * B
LOOP UNTIL A > 15
```

The following LOOP WHILE executes a block of Beep functions while A is less than or equal to 15.

```
integer A = 1, B = 1

DO
   Beep(A)
   A = (A + 1) * B
LOOP WHILE A <= 15
```

EXIT

Use the EXIT statement in a DO...LOOP or a FOR...NEXT control structure to pass control out of the current loop. EXIT takes no parameters.

Using in DO...LOOP

An EXIT statement in a DO...LOOP control structure causes control to pass to the statement following the LOOP statement. In a nested DO...LOOP structure, an EXIT statement passes control out of the *current* DO...LOOP structure.

Example

The following EXIT statement causes the loop to terminate if an element in the Nbr array equals 0.

```
int Nbr[10]
int Count = 1
// Assume values get assigned to Nbr array...

DO WHILE Count < 11
   IF Nbr[Count] = 0 THEN EXIT
   Count = Count + 1
LOOP

MessageBox("Hi",  "Count is now "  + String(Count) )
```

Using in FOR...NEXT

An EXIT statement in a FOR...NEXT control structure causes control to pass to the statement following the NEXT statement.

Example

The following EXIT statement causes the loop to terminate if an element in the Nbr array equals 0.

```
int Nbr[10]
int Count
// Assume values get assigned to Nbr array...

FOR Count = 1 to 10
   IF Nbr[Count] = 0 THEN EXIT
NEXT

MessageBox("Hi",  "Count is now "  + String(Count) )
```

FOR...NEXT

Description

The FOR...NEXT control structure is a numerical iteration. Use FOR...NEXT to execute one or more statements a specified number of times.

Syntax

```
FOR varname = start TO end {STEP increment}
    statementblock
NEXT
```

Parameter	Description
varname	The name of the iteration counter variable. It can be any numerical type (integer, double, real, long, or decimal), but integers provide the fastest performance.
start	Starting value of *varname.*
end	Ending value of *varname.*
increment	(Optional) The increment value. *Increment* must be a constant and the same data type as *varname.* If you enter an increment, STEP is required. +1 is the default increment.
statementblock	The block of statements you want to repeat.

Usage

For a positive *increment, end* must be greater than *start.* For a negative increment, end must be less than start.

When increment is positive and start is greater than end, statementblock does not execute. When increment is negative and start is less than end, statementblock does not execute.

You can nest FOR...NEXT statements. You must have a NEXT for each FOR.

A variable as the step increment
If you need to use a variable for the step increment, you can use one of the DO...LOOP constructions and increment the counter yourself within the loop.

Examples

These statements add 10 to A as long as n is >=5 and <=25.

```
FOR n = 5 to 25
   A = A+10
NEXT
```

These statements add 10 to A and increment n by 5 as long as n is >= 5 and <=25.

```
FOR N = 5 TO 25 STEP 5
   A = A+10
NEXT
```

These statements contain two lines that will never execute because *increment* is negative and *start* is less than *end.*

```
FOR Count = 1 TO 100 STEP -1
   IF Count < 1 THEN EXIT // These 2 lines
   Box[Count] = 10        // will never execute.
NEXT
```

These are nested FOR...NEXT statements.

```
Int Matrix[100,50,200]
FOR i = 1 to 100
   FOR j = 1 to 50
      FOR k = 1 to 200
         Matrix[i,j,k]=1
      NEXT
   NEXT
NEXT
```

GOTO

Description

The GOTO statement transfers control from one statement in a script to another statement that is labeled.

Syntax

GOTO *label*

Parameter	Description
label	The label associated with the statement to which you want to transfer control. A label is an identifier followed by a colon (such as OK:). Do not use the colon with a label in the GOTO statement.

Examples

The following GOTO statement skips over the Taxable=FALSE line.

```
Goto NextStep
Taxable=FALSE  //This statement will never
               //execute.
NextStep:
Rate=Count/Count4
```

The following statement transfers control to the statement associated with the label OK.

```
GOTO OK
.
.
.
OK:
.
.
.
```

HALT and RETURN

Use the HALT statement without associated keywords to terminate the application immediately. Use the RETURN statement to stop the execution of a script or function immediately.

Using HALT

Description

When PowerBuilder encounters HALT without the keyword CLOSE, it immediately terminates the application.

When PowerBuilder encounters HALT with the keyword CLOSE, it immediately executes the script for the Close event for the application and then terminates the application. If there is no script for the Close event at the application level, PowerBuilder immediately terminates the application.

Syntax

HALT {CLOSE}

Examples

In the following example, the script stops the application if the user enters a password in the SingleLineEdit named sle_password that does not match the value stored in a string named CorrectPassword.

```
IF sle_password.Text <> CorrectPassword THEN HALT
```

The following statement executes the script for the close event for the application before it terminates the application if the user enters a password in the sle_password that does not match the value stored in the string CorrectPassword.

```
IF sle_password.Text <> CorrectPassword  &
   THEN HALT CLOSE
```

Using RETURN

Description

When PowerBuilder encounters RETURN in a script, it terminates execution of that script immediately and waits for the next user action. When PowerBuilder encounters RETURN in a function, RETURN transfers (returns) control to the point at which the function was called.

Syntax

RETURN { *expression* }

Parameter	Description
expression	In a function, any value (or expression) you want the function to return. The return value must be the data type specified as the return type in the function. Do not specify an expression when you use RETURN in a script.

Examples

This script causes the system to beep once; the second beep statement will not execute.

```
Beep(1)
RETURN
Beep(1)  // This statement will not execute.
```

These statements in a user-defined function return the result of dividing Arg1 by Arg2 if Arg2 is not equal to 0; they return -1 if Arg2 is equal to 0.

```
IF Arg2 <> 0 THEN
   RETURN Arg1/Arg2
ELSE
   RETURN -1
END IF
```

IF...THEN

Use the IF...THEN control structure to cause the script to perform a specified action if a stated condition is true. IF...THEN has a single-line format and a multiline format.

Using the single-line format

Syntax

IF *condition* THEN *action1* {ELSE *action2*}

Parameter	Description
condition	The condition you want to test.
action1	The action you want performed if the condition is TRUE. The action must be a single statement on the same line as the rest of the IF statement.
action2	(Optional) The action you want performed if the condition is FALSE. The action must be a single statement on the same line as the rest of the IF statement.

You can use continuation characters to place the single-line format on more than one physical line in the script.

Examples

The following single-line IF...THEN statement opens window w_first if Num = 1; otherwise, w_rest is opened.

```
IF Num = 1 THEN Open(w_first) ELSE Open(w_rest)
```

The following single-line IF...THEN statement displays a message if the value in the SingleLineEdit sle_State is TX. It uses the continuation character to continue the single-line statement across two physical lines in the script.

```
IF sle_State.text="TX" THEN   &
   MessageBox("Hello","Tex")
```

Using the multiline format

Syntax

```
IF condition1 THEN
    action1
{ ELSEIF condition2 THEN
    action2
    . . . }
    { ELSE
    action3 }
END IF
```

Parameter	Description
condition1	The first condition you want to test.
action1	The action you want performed if *condition1* is TRUE. The action can be a statement or multiple statements that are separated by semicolons or placed on separate lines. At least one action is required.
condition2	(Optional) The condition you want to test if *condition1* is FALSE. You can have multiple ELSEIF...THEN statements in an IF...THEN control structure.
action2	The action you want performed if *condition2* is TRUE. The action can be a statement or multiple statements that are separated by semicolons or placed on separate lines.
action3	(Optional) The action you want performed if none of the preceding conditions is true. The action can be a statement or multiple statements that are separated by semicolons or placed on separate lines.

You must end a multiline IF...THEN control structure with END IF (which is two words).

Examples

The following multiline IF...THEN compares the horizontal positions of windows w_first and w_second. If w_first is to the right of w_second, w_first is moved to the left side of the screen.

```
IF w_first.X > w_second.X THEN
   w_first.X = 0
END IF
```

The following multiline IF...THEN causes the application to:

- Beep twice if X equals Y

- Display the Parts listbox and highlight item 5 if X equals Z
- Display the Choose listbox if X is blank
- Hide the Empty button and display the Full button if none of the above conditions is TRUE

```
IF X=Y THEN
   Beep(2)
ELSEIF X=Z THEN
   Show (lb_parts); lb_parts.SetState(5,TRUE)
ELSEIF X=" " THEN
   Show (lb_choose)
ELSE
   Hide(cb_empty)
   Show(cb_full)
END IF
```

CHAPTER 6

Functions

About this chapter

Much of the power of the PowerScript language resides in the built-in PowerScript functions that you can use in expressions and assignment statements. You can also extend PowerBuilder by calling external functions. This chapter describes how to use the built-in functions and how to declare external functions that reside in dynamic link libraries (DLLs).

Contents

Calling functions

To call a function, you specify the function name, followed by an open parenthesis, zero or more arguments, and a close parenthesis.

function({argument1, argument2, ...})

Most PowerScript functions require a specific number of arguments. However, some take optional arguments. The arguments can be literals, variables, other functions, or expressions.

Examples

These examples illustrate functions that take different types of arguments.

```
Now( )                       // Requires no
                             // arguments

Beep(3)                      // Requires one
                             // numeric argument

Round(123.456789, 4)         // Requires 2
                             // numeric arguments

Clipboard("PowerBuilder")    // Has one optional
                             // string argument
```

Case insensitivity

Function names are not case sensitive. For example, the following statements are equivalent:

```
Clipboard("PowerBuilder")
clipboard("PowerBuilder")
CLIPBOARD("PowerBuilder")
```

The PowerBuilder documentation shows built-in functions with uppercase letters for the first character of each word in the function name, such as MessageBox.

Naming your own functions
You can use any valid identifier (1 to 40 characters) when you name PowerScript functions that you create.

For information on user-defined functions, see the *User's Guide.*

Return values

All built-in PowerScript functions return a value. You can use the return value or ignore it.

To use the return value, assign it to a variable of the appropriate data type or call the function itself wherever you can use a value of that data type.

Examples

The built-in Asc function takes a string as an argument and returns the ASCII value of the string's first character.

```
string S1 = "Carton"
int Test
Test=32+Asc(S1)    // Test now contains the value
                   // 99 (the ASCII value of "C"
                   // is 67).
```

The SelectRow function expects a row number as the first argument. The return value of the GetRow function supplies the row number.

```
dw_1.SelectRow(dw_1.GetRow(), TRUE)
```

To ignore a return value, call the function as a single statement.

```
Beep(4)            // This returns a value, but it is
                   // rarely needed.
```

User-defined functions and external functions may or may not return a value.

How PowerBuilder looks for functions

When PowerBuilder executes a script and finds an unqualified reference to a function, it searches for the function in the following order:

1 A global external function

2 A global function

3 An object function and local external function

4 A system function

As soon as PowerBuilder finds a function with the specified name, it calls the function. If you have a global and an object function with the same name, you can call the object function by qualifying it with the object name or the pronoun This. If a function has the same name as a system function, the system function becomes inaccessible.

Types of built-in functions

The built-in PowerScript functions include object functions, which act on a instance of a particular object, and system functions, whose effects are independent of any object.

You can list all the functions in the Object browser.

❖ To list the functions:

1 Do one of the following:

- ♦ Open the PowerScript painter and click the Browse icon or select Edit➤Browse Objects from the menu bar.
- ♦ Open the Library painter and select Utilities➤Browse Objects.

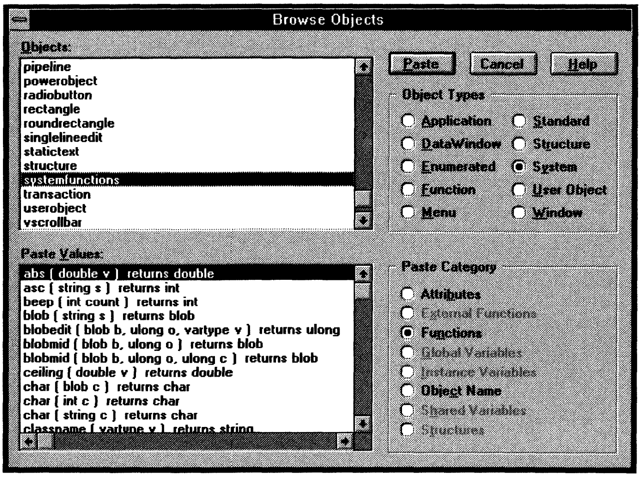
The Object browser opens.

2 Select System as the Object Type and Functions as the Paste Category.

3 In the Object listbox, select the object for which you want the list of functions.

PowerBuilder lists all the functions for the selected object.

> **Viewing the system functions**
> To see the list of system functions, choose systemfunctions in the Objects listbox.



Browse Objects

Objects:

```
pipeline
powerobject
radiobutton
rectangle
roundrectangle
singlelineedit
statictext
structure
systemfunctions
transaction
userobject
vscrollbar
```

Paste Values:

```
abs ( double v ) returns double
asc ( string s ) returns int
beep ( int count ) returns int
blob ( string s ) returns blob
blobedit ( blob b, ulong o, vartype v ) returns ulong
blobmid ( blob b, ulong o ) returns blob
blobmid ( blob b, ulong o, ulong c ) returns blob
ceiling ( double v ) returns double
char ( blob c ) returns char
char ( int c ) returns char
char ( string c ) returns char
classname ( vartype v ) returns string
```

Object Types

○ Application ○ Standard
○ DataWindow ○ Structure
○ Enumerated ● System
○ Function ○ User Object
○ Menu ○ Window

Paste Category

○ Attributes
○ External Functions
● Functions
○ Global Variables
○ Instance Variables
○ Object Name
○ Shared Variables
○ Structures

For more information

For	See
A list of PowerScript functions, organized by object type	*Objects and Controls*, which has a category for each object type and lists the functions that act on that object
A list of all functions with descriptions of their actions and arguments	*Function Reference*, which lists the functions alphabetically

Writing user-defined functions

When you need to code the same process in several scripts, in the same or different applications, you can make the code reusable by defining a user-defined function. A user-defined function is a collection of PowerScript statements that perform some processing. You can save user-defined functions in a separate library, so that any PowerBuilder application can use the functions.

👓 For information on writing user-defined functions in the Function painter, see the *User's Guide*.

External functions

External functions are functions that are written in languages other than PowerScript and stored in dynamic link libraries (DLLs). You can use external functions that are written in any language that supports the Pascal calling sequence.

Before you can use an external function in a script, you must declare it.

Two types

You can declare two types of external functions:

- **Global external functions**, which are available anywhere in the application
- **Local external functions**, which are defined for a particular type of window, menu, user object, or user-defined function. These functions are part of the object's definition and can always be used in scripts for the object itself. You can also choose to make these functions accessible to other scripts.

Syntax for declaring external functions

Use the following syntax to declare an external function.

```
{ Access } FUNCTION ReturnDataType FunctionName
( {REF} {DataType1 Arg1, ..., DataTypeN ArgN} )
LIBRARY LibName
```

You can also declare external subroutines, which are the same as external functions, except that they don't return a value.

```
{ Access } SUBROUTINE SubroutineName
( {REF} {DataType1 Arg1, ..., DataTypeN ArgN} )
LIBRARY LibName
```

Parameter	Description
Access	(Local external functions only). You can optionally specify Public, Protected, or Private to specify the access level of a local external function. The default is Public. ☞ For more information, see "Specifying access of local functions" below.
ReturnDataType	The data type of the value returned by the function.

Parameter	Description
FunctionName or *SubroutineName*	The name of a function or subroutine that resides in a DLL.
DataType1 through *DataTypeN*	The data types of the arguments (if any) specified in *Arg1* to *ArgN.*
Arg1 through *ArgN*	The names of the arguments in the function or subroutine. For more information on passing arguments, see *Building Applications.*
LibName	A string containing the name of the DLL in which the function or subroutine is stored. Microsoft Windows' DLLs usually have the extension .DLL or .EXE. Enclose the library name in quotation marks; do not include the DOS path. The library must be available to the application at execution time (see next).

Specifying access of local functions

When declaring a local external function, you can specify its **access level**—that is, you can specify which scripts have access to the function.

Access	You can use the local external function in
Public	Any script in the application.
Private	Scripts for events in the object for which the function is declared. You cannot use the function in descendants of the object.
Protected	Scripts for the object for which the function is declared and its descendants.

Access with local external functions works the same as with instance variables.

For more information about access, see the description of instance variables on page 34.

Calling local external functions

You use dot notation to call local external functions.

object.function(arguments)

For example, if you declared the local external function Reorg for the window w_emp, call the function like this.

```
w_emp.Reorg( )
```

Availability of DLL during execution

To be available to the PowerBuilder application running under Windows, the DLL must be in one of the following directories:

- The current directory
- The Windows directory
- The Windows System subdirectory
- Directories on the DOS path

Creating your own functions

When you create your own functions for use as external functions in PowerBuilder external function calls, you must create the functions using the FAR PASCAL declaration and link them in a DLL.

☞ For more information about using external functions, see *Building Applications.*

CHAPTER 7

SQL Statements

About this chapter

This chapter documents the embedded SQL and dynamic SQL statements that you can use in scripts. The first section describes using variables in SQL statements and error handling. Then the embedded SQL statements are discussed alphabetically. The last section discusses dynamic SQL.

Contents

Using SQL in scripts

PowerScript supports standard embedded SQL statements and dynamic SQL statements in scripts.

In general, PowerScript supports all DBMS-specific clauses and reserved words that occur in the supported SQL statements. For example, PowerBuilder supports DBMS-specific built-in functions within a SELECT command.

For information about embedded SQL, see online Help.

Referencing PowerScript variables in scripts

Wherever constants can be referenced in SQL statements, PowerScript variables preceded by a colon (:) can be substituted. Any valid PowerScript variable can be used.

Examples

This INSERT statement uses a constant value.

```
INSERT INTO EMPLOYEE ( SALARY )
   VALUES ( 18900 ) ;
```

The same statement using a PowerScript variable to reference the constant might look like this.

```
int   Sal_var
Sal_var = 18900
INSERT INTO EMPLOYEE ( SALARY )
   VALUES ( :Sal_var ) ;
```

Using indicator variables

PowerBuilder supports **indicator variables**, which are used to identify NULL values or conversion errors after a database retrieval. Indicator variables are integers that are specified in the *HostVariableList* of a FETCH or SELECT statement.

Each indicator variable is separated from the variable it is indicating by a space (but no comma). For example, the following statement is a *HostVariableList* without indicator variables.

```
:Name, :Address, :City
```

The same *HostVariableList* with indicator variables might look like this.

```
:Name :IndVar1, :Address :IndVar2, :City :IndVar3
```

Indicator variables have one of these values.

Numerical value	**Meaning**
0	Valid, non-NULL value
-1	NULL value
-2	Conversion error

Error reporting
Not all DBMSs return a conversion error when the data type of a column does not match the data type of the associated variable.

Examples

The following command uses the indicator variable IndVar2 to see if Address contains a NULL value.

```
if IndVar2 = -1 then...
```

You could also use the PowerScript IsNull function to accomplish the same result without using indicator variables.

```
if IsNull( Address ) then ...
```

This command uses the indicator variable IndVar3 to set City to NULL.

```
IndVar3 = -1
```

You could also use the PowerScript SetNull function to accomplish the same result without using indicator variables.

```
SetNull( City )
```

☞ For information about the SetNull function, see the *Function Reference.*

Error handling in scripts

The scripts shown in the SQL examples above do not include error handling, but it is good practice to test the success and failure codes (the SQLCode attribute) in the transaction object after *every* statement. The codes are:

Value	Meaning
0	Success.
100	The command succeeded but did not retrieve or modify any rows (which may or may not be acceptable).
-1	Error; the statement failed. Use SQLErrText or SQLDBCode to obtain the details.

About SQLErrText and SQLDBCode

The string SQLErrText in the transaction object contains the database vendor–supplied error message. The long named SQLDBCode in the transaction object contains the database vendor–supplied status code.

Example

```
IF SQLCA.SQLCode = -1 THEN
   MessageBox("SQL error", SQLCA.SQLErrText)
END IF
```

Painting standard SQL

You can paint the following SQL statements in scripts and functions:

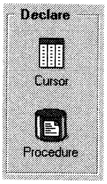
- Declarations of SQL cursors and stored procedures
- Cursor FETCH, UPDATE, and DELETE statements
- Noncursor SELECT, INSERT, UPDATE, and DELETE statements

Declaring cursors and procedures

You can declare cursors and stored procedures at the scope of global, instance, shared, or local variables.

☞ For more information about scope, see Chapter 3, "Declarations."

❖ To declare a global, instance, or shared cursor or procedure:



1 Select Declare➤Global Variables, Declare➤Instance Variables, or Declare➤Shared Variables in the Window, User Object, Menu, or PowerScript painter.

   The window that displays contains icons at the right for declaring a cursor or procedure. The Declare Cursor painter is virtually the same as the View painter, which is described in the *User's Guide.*

2 Double-click the icon and paint the statement. Supply all the required information. You can look at the SQL statement as it is being built by selecting Show SQL Syntax from the Options menu.

❖ To declare a local cursor or procedure:

1 Open the PowerScript or Function painter.

2 Click the Paste SQL button in the PainterBar, described next.

Pasting SQL statements into scripts and functions

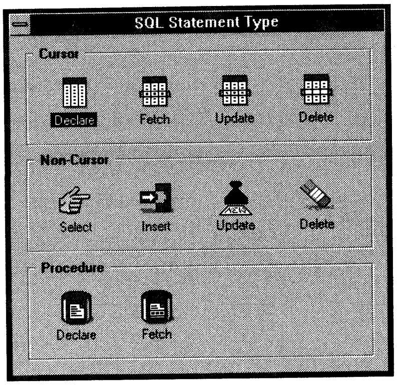
You can paint standard embedded SQL statements in the PowerScript painter, the Function painter, and the Database Administration painter.

❖ To paint an embedded SQL statement:

1 Open the PowerScript, Function, or Database Administration painter.

2 Click the Paste SQL button in the PainterBar or select Edit➤Paste SQL from the menu bar.

A window displays showing the SQL statement types that you can paint. This is the window for the PowerScript painter.



3 Select a statement type.

A window displays.

4 Create the statement by pasting and entering text, operators, and values. You can look at the SQL statement as it is being built by selecting Show SQL Syntax from the Options menu.

Supported SQL statements

In general, all DBMS-specific features are supported in PowerScript, as long as they occur within a PowerScript-supported SQL statement. For example, PowerScript supports DBMS-specific built-in functions within a SELECT command.

The rest of this chapter describes the SQL statements that PowerScript supports. The statements are listed in alphabetical order.

CLOSE Cursor

Syntax

CLOSE *CursorName* ;

Parameter	Description
CursorName	The cursor you want to close

Description

Closes the SQL cursor *CursorName*; ends processing of *CursorName*. This statement must be preceded by an OPEN statement for the same cursor. The USING TransactionObject clause is not allowed with CLOSE; the transaction object was specified in the statement that declared the cursor.

CLOSE often appears in the script that is executed when the SQL code after a fetch equals 100 (not found).

Tip
It is good practice to test the success/failure code after executing a CLOSE statement.

Example

This statement closes the Emp_cursor cursor.

```
CLOSE Emp_cursor ;
```

CLOSE Procedure

Syntax

CLOSE *ProcedureName* ;

Parameter	Description
ProcedureName	The stored procedure you want to close

DBMS-specific
Not all DBMSs support stored procedures.

Description

Closes the SQL procedure *ProcedureName*; ends processing of *ProcedureName*. This statement must be preceded by an EXECUTE statement for the same procedure. The USING TransactionObject clause is not allowed with CLOSE; the transaction object was specified in the statement that declared the procedure.

You only need to use CLOSE to close procedures that return result sets. PowerBuilder automatically closes procedures that don't return result sets (and sets the return code to 100).

CLOSE often appears in the script that is executed when the SQL code after a fetch equals 100 (not found).

Tip
It is good practice to test the success/failure code after executing a CLOSE statement.

Example

This statement closes the stored procedure named Emp_proc.

```
CLOSE Emp_proc ;
```

COMMIT

Syntax

COMMIT {USING *TransactionObject*} ;

Parameter	Description
TransactionObject	The name of the transaction object for which you want to permanently update all database operations since the previous commit, rollback, or connect. This clause is required only for transaction objects other than the default (SQLCA).

Description

Permanently updates all database operations since the previous commit, rollback, or connect for the specified transaction object. COMMIT does not cause a disconnect, but it does close all open cursors or procedures. (But note that the DISCONNECT statement in PowerBuilder does issue a COMMIT.)

> **Tip**
> It is good practice to test the success/failure code after executing a COMMIT statement.

Examples

This statement commits all operations for the database specified in the default transaction object.

```
COMMIT ;
```

This statement commits all operations for the database specified in the transaction object named Emp_tran.

```
COMMIT USING Emp_tran ;
```

CONNECT

Syntax

CONNECT {USING *TransactionObject*} ;

Parameter	Description
TransactionObject	The name of the transaction object containing the required connection information for the database to which you want to connect. This clause is required only for transaction objects other than the default (SQLCA).

Description

Connects to a specified database. This statement must be executed before any actions (such as insert, update, or delete) can be processed using the default transaction object or the specified transaction object.

> **Tip**
> It is good practice to test the success/failure code after executing a CONNECT statement.

Examples

This statement connects to the database specified in the default transaction object.

```
CONNECT ;
```

This statement connects to the database specified in the transaction object named Emp_tran.

```
CONNECT USING Emp_tran ;
```

DECLARE Cursor

Syntax

DECLARE *CursorName* CURSOR FOR *SelectStatement*
{USING *TransactionObject*} ;

Parameter	Description
CursorName	Any valid PowerBuilder name.
SelectStatement	Any valid SELECT statement.
TransactionObject	The name of the transaction object for which you want to declare the cursor. This clause is required only for transaction objects other than the default (SQLCA).

Description

Declares a cursor for the specified transaction object. DECLARE Cursor is a nonexecutable command and is analogous to declaring a variable.

To declare a global, shared, or instance cursor, select Declare➤Global Variables, Declare➤Instance Variables, or Declare➤Shared Variables in the Window, User Object, Menu, or PowerScript painter. To declare a local cursor, click the Paint SQL button in the PainterBar.

☞ For information about global, instance, shared, and local scope, see Chapter 3, "Declarations."

Example

This statement declares the cursor called Emp_cur for the database specified in the default transaction object. It also references the Sal_var variable, which must be set to an appropriate value before you execute the OPEN Emp_cur command.

```
DECLARE Emp_cur CURSOR FOR
   SELECT employee.emp_number, employee.emp_name
   FROM employee
   WHERE employee.emp_salary > :Sal_var ;
```

DECLARE Procedure

Syntax

```
DECLARE ProcedureName PROCEDURE FOR
    StoredProcedureName
    @Param1=Value1, @Param2=Value2,...
    {USING TransactionObject} ;
```

Parameter	Description
ProcedureName	Any valid PowerBuilder name.
StoredProcedureName	Any stored procedure in the database.
*@Param*n*=Value*n	The name of a parameter (argument) defined in the stored procedure and a valid PowerBuilder expression. *N* represents the number of the parameter and value.
TransactionObject	The name of the transaction object for which you want to declare the procedure. This clause is required only for transaction objects other than the default (SQLCA).

> **DBMS-specific**
> Not all DBMSs support stored procedures.

Description

Declares a procedure for the specified transaction object. DECLARE Procedure is a nonexecutable command. It is analogous to declaring a variable.

> **Using SQL Server**
> In SQL Server, you can use the optional reserved word OUT to indicate an output parameter:
>
> *@Param=Value* **OUT**

To declare a global, shared or instance procedure, select Declare➤Global Variables, Declare➤Instance Variables, or Declare➤Shared Variables in the Window, User Object, Menu, or PowerScript painter. To declare a local procedure, click the Paint SQL button in the PainterBar.

☞ For information about global, instance, shared, and local scope, see Chapter 3, "Declarations."

Example

This statement declares the procedure Emp_proc for the database specified in the default transaction object. It references the Emp_name_var and Emp_sal_var variables, which must be set to appropriate values before you execute the EXECUTE Emp_proc command.

```
DECLARE Emp_proc procedure for GetName
   @emp_name = :Emp_name_var,
   @emp_salary = :Emp_sal_var ;
```

DELETE

Syntax

DELETE FROM *TableName* WHERE *Criteria*
{USING *TransactionObject*} ;

Parameter	Description
TableName	The name of the table from which you want to delete rows.
Criteria	Criteria that specify which rows to delete.
TransactionObject	The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA).

Description

Deletes the rows in *TableName* specified by *Criteria.*

Tip
It is good practice to test the success/failure code after executing a DELETE statement.

Example

This statement deletes rows from the Employee table in the database specified in the default transaction object where Emp_num is less than 100.

```
DELETE FROM Employee WHERE Emp_num < 100 ;
```

These statements delete rows from the Employee table in the database named in the transaction object named Emp_tran where Emp_num is equal to the value entered in the SingleLineEdit sle_number.

```
int   Emp_num
Emp_num = Integer(sle_number.Text)
DELETE FROM Employee
   WHERE Employee.Emp_num = :Emp_num ;
```

The integer Emp_num requires a colon in front of it to indicate it is a variable when it is used in a WHERE clause.

DELETE Where Current of Cursor

Syntax

DELETE FROM *TableName* WHERE CURRENT OF *CursorName* ;

Parameter	Description
TableName	The name of the table from which you want to delete a row
CursorName	The name of the cursor in which the table was specified

DBMS-specific
Not all DBMSs support DELETE Where Current of Cursor.

Description

Deletes the row in which the cursor is positioned. The USING TransactionObject clause is not allowed with this form of DELETE Where Current of Cursor; the transaction object was specified in the statement that declared the cursor.

Tip
It is good practice to test the success/failure code after executing a DELETE statement.

Example

This statement deletes from the Employee table the row in which the cursor named Emp_cur is positioned.

```
DELETE FROM Employee WHERE current of Emp_curs ;
```

DISCONNECT

Syntax

DISCONNECT {USING *TransactionObject*} ;

Parameter	Description
TransactionObject	The name of the transaction object that identifies the database you want to disconnect from and in which you want to permanently update all database operations since the previous commit, rollback, or connect. This clause is required only for transaction objects other than the default (SQLCA).

Description

Executes a COMMIT for the specified transaction object, then disconnects from the specified database.

> **Tip**
> It is good practice to test the success/failure code after executing a DISCONNECT statement.

Example

This statement disconnects from the database specified in the default transaction object.

```
DISCONNECT ;
```

This statement disconnects from the database specified in the transaction object named Emp_tran.

```
DISCONNECT USING Emp_tran ;
```

EXECUTE

Syntax

EXECUTE *ProcedureName* ;

Parameter	Description
ProcedureName	The name assigned in the DECLARE statement of the stored procedure you want to execute. The procedure must have been declared previously. *ProcedureName* is not necessarily the name of the procedure stored in the database.

Description

Executes the previously declared procedure identified by *ProcedureName*. The USING TransactionObject clause is not allowed with EXECUTE; the transaction object was specified in the statement that declared the procedure.

> **Tip**
> It is good practice to test the success/failure code after executing an EXECUTE statement.

Example

This statement executes the stored procedure Emp_proc.

```
EXECUTE Emp_proc ;
```

FETCH

Syntax

FETCH *Cursor* | *Procedure* INTO *HostVariableList* ;

Parameter	Description
Cursor or *Procedure*	The name of the cursor or procedure from which you want to fetch a row
HostVariableList	PowerScript variables into which data values will be retrieved

Description

Fetches the row after the row on which *Cursor* | *Procedure* is positioned. The USING TransactionObject clause is not allowed with FETCH; the transaction object was specified in the statement that declared the cursor or procedure.

If your DBMS supports formats of FETCH other than the customary (and default) FETCH NEXT, you can specify FETCH FIRST, FETCH PRIOR, or FETCH LAST.

> **Tip**
> It is good practice to test the success/failure code after executing a FETCH statement.

Examples

This statement fetches data retrieved by the SELECT clause in the declaration of the cursor named Emp_cur and puts it into Emp_num and Emp_name.

```
int      Emp_num
string   Emp_name
FETCH Emp_cur INTO :Emp_num, :Emp_name ;
```

If sle_emp_num and sle_emp_name are SingleLineEdits, these statements fetch from the cursor named Emp_cur, store the data in Emp_num and sle_emp_name, and then convert Emp_num from an integer to a string and put it in sle_emp_num.

```
int      Emp_num
FETCH Emp_cur into :emp_num, :sle_emp_name.Text ;
sle_emp_num.Text = string(Emp_num)
```

INSERT

Syntax

INSERT *RestOfInsertStatement*
{USING *TransactionObject*} ;

Parameter	Description
RestOfInsertStatement	The rest of the INSERT statement (the INTO clause, list of columns and values or source).
TransactionObject	The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA).

Description

Inserts one or more new rows into the table specified in *RestOfInsertStatement.*

Tip
It is good practice to test the success/failure code after executing an INSERT statement.

Examples

These statements insert a row with the values in EmpNbr and EmpName into the Emp_nbr and Emp_name columns of the Employee table identified in the default transaction object.

```
int      EmpNbr
string   EmpName
...
INSERT INTO Employee (employee.Emp_nbr,
   employee.Emp_name)
   VALUES (:EmpNbr, :EmpName) ;
```

These statements insert a row with the values entered in the SingleLineEdits sle_number and sle_name into the Emp_nbr and Emp_name columns of the Employee table in the transaction object named Emp_tran.

```
int      EmpNbr
EmpNbr = Integer(sle_number.Text)
INSERT INTO Employee (employee.Emp_nbr,
   employee.Emp_name)
   VALUES (:EmpNbr, :sle_name.Text) USING Emp_tran ;
```

OPEN Cursor

Syntax

OPEN *CursorName* ;

Parameter	Description
CursorName	The name of the cursor you want to open

Description

Causes the SELECT specified when the cursor was declared to be executed. The USING TransactionObject clause is not allowed with OPEN; the transaction object was specified in the statement that declared the cursor.

> **Tip**
> It is good practice to test the success/failure code after executing an OPEN statement.

Example

This statement opens the cursor Emp_curs.

```
OPEN Emp_curs ;
```

ROLLBACK

Syntax

ROLLBACK {USING *TransactionObject*} ;

Parameter	Description
TransactionObject	The name of the transaction object that identifies the database in which you want to cancel all operations since the last commit, rollback, or connect. This clause is required only for transaction objects other than the default (SQLCA).

Description

Cancels all database operations in the specified database since the last COMMIT, ROLLBACK, or CONNECT. ROLLBACK does not cause a disconnect, but it does close all open cursors and procedures.

> **Tip**
> It is good practice to test the success/failure code after executing a ROLLBACK statement.

Examples

This statement cancels all database operations in the database specified in the default transaction object.

```
ROLLBACK ;
```

This statement cancels all database operations in the database specified in the transaction object named Emp_tran.

```
ROLLBACK USING emp_tran ;
```

SELECT

Syntax

SELECT *RestOfSelectStatement*
    {USING *TransactionObject*} ;

Parameter	Description
RestOfSelectStatement	The rest of the SELECT statement (the column list INTO, FROM, WHERE, and other clauses).
TransactionObject	The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA).

Description

Selects a row in the tables specified in *RestOfSelectStatement.*

An error occurs if the SELECT statement returns more than one row.

> **Tip**
> It is good practice to test the success/failure code after executing a SELECT statement.

Example

The following statements select data in the Emp_LName and Emp_FName columns of a row in the Employee table and put the data into the SingleLineEdits sle_LName and sle_FName. The transaction object Emp_tran is used.

```
int   Emp_num
Emp_num = Integer(sle_Emp_Num.Text)

SELECT employee.Emp_LName, employee.Emp_FName
   INTO :sle_LName.text, :sle_FName.text
   FROM Employee
   WHERE Employee.Emp_nbr = :Emp_num
   USING Emp_tran ;

if Emp_tran.SQLCode = 100 then
   MessageBox("Employee Inquiry", &
      "Employee Not Found")
elseif Emp_tran.SQLCode > 0 then
   MessageBox("Database Error", &
      Emp_tran.SQLErrText, Exclamation!)
End If
```

SELECTBLOB

Syntax

SELECTBLOB *RestOfSelectStatement*
{USING *TransactionObject*} ;

Parameter	Description
RestOfSelectStatement	The rest of the SELECT statement (the INTO, FROM, and WHERE clauses).
TransactionObject	The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA).

You can an include indicator variable in the host variable list (target parameters) in the INTO clause to check for an empty blob (a blob of 0 length) and conversion errors.

Description

Selects a single blob column in a row in the table specified in *RestOfSelectStatement.*

An error occurs if the SELECTBLOB statement returns more than one row.

> **Tip**
> It is good practice to test the success/failure code after executing a SELECTBLOB statement.

Example

The following statements select the blob column Emp_pic from a row in the Employee table and set the picture p_1 to the bitmap in Emp_id_pic. The transaction object Emp_tran is used.

```
Blob     Emp_id_pic
SELECTBLOB Emp_pic
   INTO  :Emp_id_pic
   FROM Employee
   WHERE Employee.Emp_Num = 100
   USING Emp_tran ;
p_1.SetPicture(Emp_id_pic)
```

The blob Emp_id_pic requires a colon to indicate it is a host (PowerScript) variable when you use it in the INTO clause of the SELECTBLOB statement.

UPDATE

Syntax

UPDATE *TableName RestOfUpdateStatement*
 {USING *TransactionObject*} ;

Parameter	Description
TableName	The name of the table in which you want to update rows.
RestOfUpdateStatement	The rest of the UPDATE statement (the SET and WHERE clauses).
TransactionObject	The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA).

Description

Updates the rows specified in *RestOfUpdateStatement.*

Tip
It is good practice to test the success/failure code after executing a UPDATE statement.

Example

These statements update rows from the Employee table in the database specified in the transaction object named Emp_tran where Emp_num is equal to the value entered in the SingleLineEdit sle_Number.

```
int   Emp_num
Emp_num=Integer(sle_Number.Text )
UPDATE Employee
   SET emp_name = :sle_Name.Text
   WHERE Employee.emp_num  = :Emp_num
   USING Emp_tran ;
```

The integer Emp_num and the SingleLineEdit sle_name require a colon to indicate they are host (PowerScript) variables when you use them in an UPDATE statement.

UPDATEBLOB

Syntax

UPDATEBLOB *TableName*
    SET *BlobColumn* = *BlobVariable*
    RestOfUpdateStatement {USING *TransactionObject*} ;

Parameter	Description
TableName	The name of the table you want to update.
BlobColumn	The name of the column you want to update in *TableName*. The data type of this column must be blob.
BlobVariable	A PowerScript variable of the data type blob.
RestOfUpdateStatement	The rest of the UPDATE statement (the WHERE clause).
TransactionObject	The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA).

Description

Updates the rows in *TableName* in *BlobColumn*.

> **Tip**
> It is good practice to test the success/failure code after executing a UPDATEBLOB statement.

Example

These statements update the blob column emp_pic in the Employee table where emp_num is 100.

```
int   fh
blob  Emp_id_pic
fh = FileOpen("c:\emp_100.bmp", StreamMode!)
IF fh <> -1 THEN
   FileRead(fh, emp_id_pic)
   FileClose(fh)
   UPDATEBLOB Employee SET emp_pic = :Emp_id_pic
      WHERE Emp_num = 100
      USING Emp_tran ;
END IF
```

The blob Emp_id_pic requires a colon to indicate it is a host (PowerScript) variable in the UPDATEBLOB statement.

UPDATE Where Current of Cursor

Syntax

UPDATE *TableName SetStatement*
WHERE CURRENT OF *CursorName* ;

Parameter	Description
TableName	The name of the table in which you want to update the row
SetStatement	The word SET followed by a comma-separated list of the form *ColumnName = value*
CursorName	The name of the cursor in which the table is referenced

Description

Updates the row in which the cursor is positioned using the values in *SetStatement.* The USING Transaction Object clause is not allowed with UPDATE Where Current of Cursor; the transaction object was specified in the statement that declared the cursor.

Example

This statement updates the row in the Employee table in which the cursor called Emp_curs is positioned.

```
UPDATE Employee
   SET salary = 17800
   WHERE CURRENT of Emp_curs ;
```

Using dynamic SQL

Database applications usually perform a specific activity, so you usually know the complete SQL statement when you write and compile the script. When PowerBuilder does not support the statement in embedded SQL (for example, a DDL statement) or when the parameters or the format of the statements are unknown at compile time, the application must build the SQL statements at execution time. This is called **dynamic SQL**. The parameters used in dynamic SQL statements can change each time the program is executed.

Using WATCOM SQL
For information about using dynamic SQL with WATCOM SQL, see *WATCOM SQL.*

Four formats of dynamic SQL

PowerBuilder has four dynamic SQL formats. Each format handles one of the following situations at compile time.

Format	When used
Format 1	Non-result-set statements with no input parameters
Format 2	Non-result-set statements with input parameters
Format 3	Result-set statements in which the input parameters and result-set columns are known at compile time
Format 4	Result set statements in which the input parameters, the result-set columns, or both, are unknown at compile time

To handle these situations, use:

- The PowerBuilder dynamic SQL statements
- The dynamic versions of CLOSE, DECLARE, FETCH, OPEN, and EXECUTE
- The PowerBuilder data types DynamicStagingArea and DynamicDescriptionArea

The syntax for each situation follows, with examples.

About the examples
The examples assume that the default transaction object (SQLCA) has been assigned valid values and that a successful CONNECT has been executed. Although the examples do not show error checking, you should check the SQLCode after each SQL statement.

PowerBuilder's dynamic SQL statements

The dynamic SQL statements are:

```
DESCRIBE DynamicStagingArea
INTO DynamicDescriptionArea ;

EXECUTE {IMMEDIATE} SQLStatement
{USING TransactionObject} ;

EXECUTE DynamicStagingArea
USING ParameterList;

EXECUTE DYNAMIC Cursor | Procedure
USING ParameterList ;

OPEN DYNAMIC Cursor | Procedure
USING ParameterList ;

EXECUTE DYNAMIC Cursor | Procedure
USING DESCRIPTOR DynamicDescriptionArea ;

OPEN DYNAMIC Cursor | Procedure
USING DESCRIPTOR DynamicDescriptionArea ;

PREPARE DynamicStagingArea
FROM SQLStatement {USING TransactionObject} ;
```

About DynamicStagingArea

DynamicStagingArea is a PowerBuilder data type. PowerBuilder uses a variable of this type to store information for use in subsequent statements.

The DynamicStagingArea is the only connection between the execution of a statement and a transaction object and is used internally by PowerBuilder; you cannot access information in the DynamicStagingArea.

PowerBuilder provides a global DynamicStagingArea variable named SQLSA that you can use when you need a DynamicStagingArea variable. If necessary, you can declare and create additional variables of this type.

After the EXECUTE statement is completed, SQLSA is no longer referenced.

About DynamicDescriptionArea

DynamicDescriptionArea is a PowerBuilder data type. PowerBuilder uses a variable of this type to store information about the input and output parameters used in Format 4 of dynamic SQL.

PowerBuilder provides a global DynamicDescriptionArea named SQLDA that you can use when you need a DynamicDescriptionArea variable. If necessary, you can declare and create additional variables of this type.

👓 For more information about SQLDA, see "Format 4" on page 128.

Format 1

Use this format to execute a SQL statement that does not produce a result set and does not require input parameters. You can use this format to execute all forms of Data Definition Language (DDL).

Syntax

EXECUTE IMMEDIATE *SQLStatement*
{USING *TransactionObject*} ;

Parameter	Description
SQLStatement	A string containing a valid SQL statement. The string can be a string constant or a PowerBuilder variable preceded by a colon (such as :mysql). The string must be contained on one line and cannot contain expressions.
TransactionObject	The name of the transaction object that identifies the database.

Examples

This statement creates a database table named Employee. The statements use the string Mysql to store the CREATE statement.

> **For SQL Server users**
> If you are connected to a SQL Server database, set AUTOCOMMIT to TRUE before executing the CREATE.

```
string  Mysql
Mysql = "CREATE TABLE Employee "&
   +"(emp_id integer not null,"&
   +"dept_id integer not null, "&
   +"emp_fname char(10) not null, "&
   +"emp_lname char(20) not null)"
EXECUTE IMMEDIATE :Mysql ;
```

This statement assumes a transaction object named My_trans exists and is connected.

```
string  Mysql
Mysql="INSERT INTO dept Values (1234, 'Purchasing')"
EXECUTE IMMEDIATE :Mysql USING My_trans ;
```

Format 2

Use this format to execute a SQL statement that does not produce a result set but does require input parameters. You can use this format to execute all forms of Data Definition Language (DDL).

Syntax

PREPARE *DynamicStagingArea* FROM *SQLStatement*
{USING *TransactionObject*} ;

EXECUTE *DynamicStagingArea*
USING {*ParameterList*} ;

Parameter	Description
DynamicStagingArea	The name of the DynamicStagingArea (usually SQLSA).
SQLStatement	A string containing a valid SQL statement. The string can be a string constant or a PowerBuilder variable preceded by a colon (such as :mysql). The string must be contained on one line and cannot contain expressions. Enter a question mark (?) for each parameter in the statement. Value substitution is positional; reserved word substitution is not allowed.
TransactionObject	The name of the transaction object that identifies the database.
ParameterList	A comma-separated list of PowerScript variables. Note that PowerScript variables are preceded by a colon (:).

Description

To specify a NULL value, use the SetNull function.

Examples

These statements prepare a DELETE statement with one parameter in SQLSA, then execute it using the value of the PowerScript variable Emp_id_var.

```
INT   Emp_id_var = 56
PREPARE SQLSA
   FROM "DELETE FROM employee WHERE emp_id=?" ;
EXECUTE SQLSA USING :Emp_id_var ;
```

These statements prepare an INSERT statement with two parameters in SQLSA, then execute it using the value of the PowerScript variables Dept_id_var and Dept_name_var. Note that Dept_name_var is NULL.

```
INT      Dept_id_var = 156
String   Dept_name_var
SetNull(Dept_name_var)
PREPARE SQLSA
   FROM "INSERT INTO dept VALUES (?,?)" ;
EXECUTE SQLSA USING :Dept_id_var,:Dept_name_var ;
```

Format 3

Use this format to execute a SQL statement that produces a result set in which the input parameters and result set columns are known at compile time.

Syntax

```
DECLARE Cursor | Procedure
    DYNAMIC CURSOR | PROCEDURE
    FOR DynamicStagingArea ;

PREPARE DynamicStagingArea FROM SQLStatement
    {USING TransactionObject} ;

OPEN DYNAMIC Cursor
    {USING ParameterList} ;

EXECUTE DYNAMIC Procedure
    {USING ParameterList} ;

FETCH Cursor | Procedure
    INTO HostVariableList} ;

CLOSE Cursor | Procedure ;
```

Parameter	Description
Cursor or *Procedure*	The name of the cursor or procedure you want to use.
DynamicStagingArea	The name of the DynamicStagingArea (usually SQLSA).
SQLStatement	A string containing a valid SQL SELECT statement. The string can be a string constant or a PowerBuilder variable preceded by a colon (such as :mysql). The string must be contained on one line and cannot contain expressions. Enter a question mark (?) for each parameter in the statement. Value substitution is positional; reserved word substitution is not allowed.
TransactionObject	The name of the transaction object that identifies the database.

Parameter	Description
ParameterList	A comma-separated list of PowerScript variables. Note that PowerScript variables are preceded by a colon (:).
HostVariableList	The list of PowerScript variables into which the data values will be retrieved.

Description

To specify a NULL value, use the SetNull function.

The DECLARE statement is not executable and can be declared globally.

If your DBMS supports formats of FETCH other than the customary (and default) FETCH NEXT, you can specify FETCH FIRST, FETCH PRIOR, or FETCH LAST.

The FETCH and CLOSE statements in Format 3 are the same as in standard embedded SQL.

To declare a global, shared, or instance cursor or procedure, select Global Variables, Instance Variables, or Shared Variables on the Declare menu of the PowerScript painter. To declare a local cursor, click the Paint SQL button in the PainterBar.

For information about global, instance, shared, and local scope, see Chapter 3, "Declarations."

Examples

The statements in this example associate a cursor named my_cursor with SQLSA, prepare a SELECT statement in SQLSA, open the cursor, and return the employee ID in the current row into the PowerScript variable Emp_id_var.

```
INT   Emp_id_var
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
PREPARE SQLSA FROM "SELECT emp_id FROM employee" ;
OPEN DYNAMIC my_cursor ;
FETCH my_cursor INTO :Emp_id_var ;
CLOSE my_cursor ;
```

You can loop through the cursor as you can in embedded static SQL.

The statements in this example associate a cursor named my_cursor with SQLSA, prepare a SELECT statement with one parameter in SQLSA, open the cursor, and substitute the value of the variable Emp_state_var for the parameter in the SELECT statement. The employee ID in the active row is returned into the PowerBuilder variable Emp_id_var.

```
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
INT      Emp_id_var
String   Emp_state_var = "MA"
String   Sqlstatement

Sqlstatament = "SELECT emp_id FROM employee "&
   +"WHERE emp_state = ?"
PREPARE SQLSA FROM :Sqlstatement ;
OPEN DYNAMIC my_cursor using :Emp_state_var ;
FETCH my_cursor INTO :Emp_id_var ;
CLOSE my_cursor ;
```

The statements in this example perform the same processing as the preceding example but use a database stored procedure called Emp_select.

```
// The syntax of emp_select is:
// "SELECT emp_id
// FROM employee WHERE emp_state=@stateparm".
DECLARE my_proc DYNAMIC PROCEDURE FOR SQLSA ;
INT      Emp_id_var
String   Emp_state_var

PREPARE SQLSA FROM "emp_select @stateparm=?" ;
Emp_state_var = "MA"
EXECUTE DYNAMIC my_proc USING :Emp_state_var ;
FETCH my_proc INTO :Emp_id_var ;
CLOSE my_proc ;
```

Format 4

Use this format to execute a SQL statement that produces a result set in which the number of input parameters, or the number of result-set columns, or both, are unknown at compile time.

Syntax

```
DECLARE Cursor | Procedure
    DYNAMIC CURSOR | PROCEDURE
    FOR DynamicStagingArea ;

PREPARE DynamicStagingArea FROM SQLStatement
    {USING TransactionObject} ;

DESCRIBE DynamicStagingArea
    INTO DynamicDescriptionArea ;

OPEN DYNAMIC Cursor | Procedure
    USING DESCRIPTOR DynamicDescriptionArea} ;

EXECUTE DYNAMIC Cursor | Procedure
    USING DESCRIPTOR DynamicDescriptionArea ;

FETCH Cursor | Procedure
    USING DESCRIPTOR DynamicDescriptionArea ;

CLOSE Cursor | Procedure ;
```

Parameter	Description
Cursor or *Procedure*	The name of the cursor or procedure you want to use.
DynamicStagingArea	The name of the DynamicStagingArea (usually SQLSA).
SQLStatement	A string containing a valid SQL SELECT statement. The string can be a string constant or a PowerBuilder variable preceded by a colon (such as :mysql). The string must be contained on one line and cannot contain expressions. Enter a question mark (?) for each parameter in the statement. Value substitution is positional; reserved word substitution is not allowed.

Parameter	Description
TransactionObject	The name of the transaction object that identifies the database.
DynamicDescriptionArea	The name of the DynamicDescriptionArea (usually SQLDA).

Description

The DECLARE statement is not executable and can be defined globally.

If your DBMS supports formats of FETCH other than the customary (and default) FETCH NEXT, you can specify FETCH FIRST, FETCH PRIOR, or FETCH LAST.

To declare a global, shared, or instance cursor or procedure, select Global Variables, Instance Variables, or Shared Variables on the Declare menu of the PowerScript painter. To declare a local cursor, click the Paint SQL button in the PainterBar.

☞ For information about global, instance, shared, and local scope, see Chapter 3, "Declarations."

Accessing attribute information

When a statement is described into a DynamicDescriptionArea, the information in the following table is available to you in the NumInputs, InParmType, NumOutputs, and OutParmType attributes of that DynamicDescriptionArea variable.

Information	Attribute
Number of input parameters	NumInputs
Array of input parameter types	InParmType
Number of output parameters	NumOutputs
Array of output parameter types	OutParmType

The array of input parameter values and the array of output parameter values are also available. You can use the SetDynamicParm function to set the values of an input parameter and the following functions to obtain the value of an output parameter:

- GetDynamicDate
- GetDynamicDateTime
- GetDynamicNumber
- GetDynamicString

- GetDynamicTime

☞ For information about these functions, see the *Function Reference.*

Parameter values

The following enumerated data types are the valid values for the input and output parameter types:

TypeBoolean!

TypeDate!

TypeDateTime!

TypeDecimal!

TypeDouble!

TypeInteger!

TypeLong!

TypeReal!

TypeString!

TypeTime!

TypeUnsignedInteger!

TypeUnsignedLong!

Input parameters

You can set the type and value of each input parameter found in the PREPARE statement. PowerBuilder populates the SQLDA attribute NumInputs when the DESCRIBE is executed. You can use this value with the SetDynamicParm function to set the type and value of a specific input parameter. The input parameters are optional. However, if you use them, you should fill in all the values before executing the OPEN or EXECUTE statement.

Output parameters

You can access the type and value of each output parameter found in the PREPARE statement. If the database supports output parameter description, PowerBuilder populates the SQLDA attribute NumOutputs when the DESCRIBE is executed. If the database does not support output parameter description, PowerBuilder populates the SQLDA attribute NumOutputs when the FETCH statement is executed.

You can use the number of output parameters in the NumOutputs attribute in functions to obtain the type of a specific parameter from the output parameter type array in the OutParmType attribute. When you have the type, you can call the appropriate function after the FETCH statement to retrieve the output value.

Examples

This example assumes you know that there will be only one output descriptor and that it will be an integer. You can expand this example to support any number of output descriptors and any data type by wrapping the CHOOSE CASE statement in a loop and expanding the CASE statements.

```
string   Stringvar, Sqlstatement
int      Intvar
Sqlstatement = "SELECT emp_id FROM employee"
PREPARE SQLSA FROM :Sqlstatement ;
DESCRIBE SQLSA INTO SQLDA ;
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
OPEN DYNAMIC my_cursor USING DESCRIPTOR SQLDA ;
FETCH my_cursor USING DESCRIPTOR SQLDA ;

// If the FETCH is successful, the output
// descriptor array will contain returned
// values from the first row of the result set.
// SQLDA.NumOutputs contains the number of
// output descriptors.

// The SQLDA.OutParmType array will contain
// NumOutput entries and each entry will contain
// an value of the enumerated data type ParmType
// (such as TypeInteger!, or TypeString!).

CHOOSE CASE SQLDA.OutParmType[1]
   CASE TypeString!
      Stringvar = GetDynamicString(SQLDA, 1)
   CASE TypeInteger!
      Intvar = GetDynamicNumber(SQLDA, 1)
END CHOOSE
CLOSE my_cursor ;
```

This example assumes you know there is one string input descriptor and sets the parameter to MA.

```
string   Sqlstatement
Sqlstatement = "SELECT emp_id FROM employee "&
   +"WHERE emp_state = ?"
PREPARE SQLSA FROM :Sqlstatement ;
DESCRIBE SQLSA INTO SQLDA ;
// If the DESCRIBE is successful, the input
// descriptor array will contain one input
// descriptor that you must fill prior to the OPEN
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
SetDynamicParm(SQLDA, 1, "MA")
OPEN DYNAMIC my_cursor USING DESCRIPTOR SQLDA ;
FETCH my_cursor USING DESCRIPTOR SQLDA ;
// If the FETCH is successful, the output
// descriptor array will contain returned
// values from the first row of the result set
// as in the first example.
CLOSE my_cursor ;
```

Considerations

When you use dynamic SQL, you must:

- Prepare the DynamicStagingArea in all formats except Format 1
- Describe the DynamicDescriptionArea in Format 4
- Execute the statements in the appropriate order
- Understand how the Where Current of Cursor clause works

Preparation and description

Since the SQLSA staging area is the only connection between the execution of a SQL statement and a transaction object, an execution error will occur if you do not prepare the SQL statement correctly.

In addition to SQLSA and SQLDA, you can declare other variables of the DynamicStagingArea and DynamicDescriptionArea data types. However, this is required *only* when your script requires simultaneous access to two or more dynamically prepared statements.

Examples

This is a valid dynamic cursor.

```
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
PREPARE SQLSA FROM "SELECT emp_id FROM employee" ;
OPEN DYNAMIC my_cursor ;
```

This is an invalid dynamic cursor. There is no PREPARE, and therefore an execution error will occur.

```
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
OPEN DYNAMIC my_cursor ;
```

Statement order

Where you place the statements in your scripts is unimportant, but the order of execution is important in Formats 2, 3, and 4. You must execute:

- The DECLARE and the PREPARE before you execute any other dynamic SQL statements
- The OPEN in Formats 3 and 4 before the FETCH
- The CLOSE at the end

If you have multiple PREPARE statements, the order affects the contents of SQLSA.

Example

These statements illustrate the correct ordering.

```
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA
string sql1, sql2
sql1 = "SELECT emp_id FROM department "&
WHERE salary > 90000"
sql2 = "SELECT emp_id FROM department "&
WHERE salary > 20000"

IF deptId = 200 then
   PREPARE SQLSA FROM :sql1 USING SQLCA ;
ELSE
   PREPARE SQLSA FROM :sql2 USING SQLCA ;
END IF
OPEN DYNAMIC my_cursor ; // my_cursor maps to the
                         // SELECT that has been
                         // prepared.
```

Using Where Current Of

The Where Current Of Cursor clause works with dynamically created cursors, but its execution is not dynamic.

Therefore, you should *not* try to execute statements like this.

```
UPDATE EMP SET EMP_STATE = 'CT'
   WHERE CURRENT OF my_cursor ;
```

These statements are valid.

```
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
PREPARE SQLSA FROM "SELECT * FROM employee"
   USING SQLCA ;
OPEN DYNAMIC my_cursor ;
FETCH my_cursor INTO :var1, :var2 ;
UPDATE employee SET emp_state = 'CT'
   WHERE CURRENT OF my_cursor ;
```

APPENDIX A

PowerBuilder Units

About this appendix

PowerBuilder units are used to define the x and y coordinate positions and the width and height of a window and all controls in the window.

Contents

Benefits of PowerBuilder units

The benefits of PowerBuilder units include:

- A window and all its controls are reproduced exactly (pixel for pixel) when run at a later time on the same machine.
- A window designed on one machine is reproduced exactly (pixel for pixel) on any machine with the same type of monitor (such as VGA or EGA) and the same system font.
- Screens designed on one machine (for example, one with a VGA display and a 16-pixel system font) and run on another (for example, one with an EGA display and a 12-pixel system font) are very similar.

How PowerBuilder units are calculated

PowerBuilder units are based on the system font (the font Windows uses for captions, menus, or listboxes). This is the same method used by Windows for dialog boxes, where sizes are defined in terms of 1/4 the character width and 1/8 the character height. However, these Windows dialog box units are not granular enough to define sizes or positions less than two pixels on a VGA or EGA screen.

PowerBuilder units, on the other hand, provide eight times greater resolution and can reproduce one-pixel dimensions on even high-resolution (for example, 2048x2048) monitors. Specifically, a horizontal unit is 1/32 the width of an average character in the system font (tmAveCharWidth), and a vertical unit is 1/64 the system font height (tmHeight).

Sizes in the Window painter and in scripts are in PowerBuilder units. In fact, you rarely see or use pixel measurements. (The one exception is the grid size in the Window and DataWindow painters, which is in pixels.)

Converting between PowerBuilder units and pixels

Internally, PowerBuilder uses the following formulas for converting between PowerBuilder units and pixels. PowerBuilder calculates the formulas using integer arithmetic, so all fractional values are dropped after each step of the calculation.

- For x coordinate locations and object widths:

```
units = (64 * pixels / ( 2 * FontWidth)) + 1
pixels = 2 * units * FontWidth / 64
```

- For y coordinate locations and object heights:

```
units = (128 * pixels / (2 * FontHeight)) + 1
pixels = 2 * units * FontHeight / 128
```

Although PowerBuilder units for x coordinate locations and widths are 1/32 the width of an average character in the system font, these formulas use 64 and two times the system font width. This gives the same result and makes it possible to use integer arithmetic. Integer arithmetic truncates remainders (if any) at each step, and this truncation is an essential part of each calculation.

Similarly, for y coordinate locations and heights, the formulas use 128 and two times the system font height to calculate units that are 1/64 of the system font height.

Examples of conversions

Suppose you have a ListBox located at x=50, y=100 (pixels) with a width of 150 and a height of 200, and you are on a VGA screen (640x480 pixels) where the system font height is 16 pixels and its width is seven pixels. The following table shows how the pixel measurements would be converted automatically to PowerBuilder units.

Coordinate/ dimension	Pixels	PowerBuilder units
x	50	229
y	100	401
Width	150	686
Height	200	801

The following table shows the results when these units are converted back to pixels for display on the same machine.

Coordinate/ dimension	PowerBuilder units	Pixels	How close?
x	229	50	Identical
y	401	100	Identical
Width	686	150	Identical
Height	801	200	Identical

The following table shows the results when these units are converted on an EGA monitor with a system font height of 12 pixels.

Coordinate/ dimension	PowerBuilder Units	Pixels	How close?
x	229	50	Identical
y	401	75	EGA is 21.4 % of the screen height; VGA is 20.8 %
Width	686	150	Identical
Height	801	150	EGA is 42.9 % of the screen height; VGA is 41.7 %

Explanation

When developing on VGA and deploying on EGA, the horizontal measurements are identical, because both have 640 pixels across and all the standard system fonts are seven pixels average width. You get the closest match in vertical dimensions when you have a 16-pixel system font on the VGA and a 12-pixel system font on the EGA, as shown in the following table.

Match	VGA system font height	EGA system font height	Maximum error
Best	16	12	3 %
	15	10	9 %
	15	12	10 %
Worst	16	10	14 %

Additional factors

On most systems, squares (and circles) do not have the same height and width measured in PowerBuilder units. The easiest and most accurate way to draw a square is to set the grid size (which is measured in pixels) in the Window painter. On a VGA (640x480 pixels), there are the same number of pixels per inch horizontally and vertically so that making the grid size the same in both dimensions helps to produce accurate squares. On an EGA (640x350), you should set the vertical grid size to 73 percent (350/480) of the horizontal grid size. For example, use seven for the horizontal grid and five for the vertical grid.

On a system with a large-screen monitor, a window that fills a VGA screen often does not fill the entire large screen. This is deliberate. The window size, text, and controls are in the correct proportions and are at least as legible as on a VGA screen. For example, on a 1664x1200 monitor with a 24-pixel system font, a window that filled a VGA screen takes about 77 percent of the screen's width and 60 percent of the screen's height.

Conversion functions

PowerScript provides the following functions to convert between PowerBuilder units and pixels.

Function	Returned data type	Use to
PixelsToUnits	Integer	Convert pixels to PowerBuilder units
UnitsToPixels	Integer	Convert PowerBuilder units to pixels

APPENDIX B

Reserved Words

You cannot use the following reserved words as identifiers because PowerScript uses them internally.

and
call
case
choose
close
commit
connect
continue
create
cursor
declare
delete
describe
descriptor
destroy
disconnect
do
dynamic
else
elseif
end
enumerated
event
execute
exit
external
false
fetch
first
for
forward
from
function
global
goto
halt
if
immediate
insert
into
is
last
library
loop
next
not
of
on
open
or
parent
prepare
prior
private
procedure
protected
prototypes
public
readonly
ref
return
rollback
select
selectblob
shared
step
subroutine
super
system
then
this
to
true
type
until
update
updateable
using
variables
while
with
within

APPENDIX C

Supported C Data Types

The following table lists the PowerBuilder-supported C data types and their PowerBuilder equivalents.

C data type	PowerBuilder equivalent	Description
UNSIGNED	UINT	16-bit unsigned integer
LONG	LONG	32-bit signed integer
BYTE	CHAR	8-bit unsigned character
CHAR	CHAR	8-bit unsigned character
BOOL	BOOLEAN	16-bit signed integer
WORD	UINT	16-bit unsigned integer
DWORD	ULONG	32-bit unsigned integer
LPSTR	STRING	32-bit far pointer to a character string
LPBYTE	STRING	32-bit far pointer to a character
LPINT	STRING	32-bit far pointer to an integer
LPWORD	STRING	32-bit far pointer to an unsigned integer
LPLONG	STRING	32-bit far pointer to a long
LPDWORD	STRING	32-bit far pointer to a double word
LPVOID	STRING	32-bit far pointer to any data type
HANDLE	UINT	16-bit handle to a Windows object (for example, HICON or HBITMAP)

The C near-pointer data types (such as PSTR and NPSTR) are not supported in PowerBuilder.

Use the REF reserved word in external function declarations that require a 32-bit far pointer to a PowerBuilder variable. For example, to prototype a C function called MyFunc, enter.

```
BOOL FAR PASCAL MyFunc(HANDLE FAR *lpHandle);
```

To call MyFunc from PowerBuilder, declare it as follows.

```
FUNCTION boolean MyFunc(REF UINT lpHandle)  &
   LIBRARY "myfuncs.dll"
```

After you declare the function, you can call it as follows.

```
UINT hWnd               // A handle to a window.
IF MyFunc(hWnd) THEN    // Function succeeded,
                        // caller filled in handle.
ELSE                    // Function failed.
END IF
```

PowerBuilder will pass the internal memory address of the variable hWnd so the called function can fill in the value. This is known as being *passed by reference.*

APPENDIX D

Floating-Point Limits by Platform

The following table lists the various platforms that PowerBuilder supports and the limits for real and double values.

Platform and data type	Minimum	Maximum
Macintosh		
Double	2.225074E-308	1.797693E+308
Real	1.17549E-38	3.402823E+38
UNIX (Sun SPARC and HP PA-RISC)		
Double	2.2250738585072014E-308	1.7976931348623157E+308
Real	1.17549435E-38	3.40282347E+38
Windows 3.1		
Double	2.225073858507202E-308	1.797693134862315E+308
Real	1.175494351E-38	3.402823466E+38
Windows NT		
Double	2.2250738585072014E-308	1.7976931348623158E+308
Real	1.175494351E-38	3.402823466E+38

Index