Inscribe User's Guide

Pervasive Software Inc.
8834 Capital of Texas Highway
Austin, Texas 78759 USA

PERVASIVE
SOFTWARE

Inscribe User’s Guide 100-003246-004 February 1998

Contents

Figures

Tables

About This Manual

This manual is intended for software developers using Inscribe to develop scripts and integrate them with Pervasive Software’s database applications. Inscribe uses the Softbridge Basic Language (SBL), also described in this manual. You use Inscribe and SBL to create scripts that automate a variety of daily tasks.

Pervasive Software would appreciate your comments and suggestions about this manual. Please complete the User Comments form that appears at the back, and fax or mail it to Pervasive Software, or send email to docs@pervasive.com.

Organization

The following list briefly describes each chapter and appendix in the manual:

- Chapter 1—“Introduction”

  This chapter introduces the Inscribe Interface Engine and Developer Kit.

- Chapter 2—“Language Overview”

  This chapter describes the essential rules and components of SBL.

- Chapter 3—“Statements and Functions”

  This chapter provides a quick reference for the SBL statements and functions.

- Appendix A—“SBL and Visual Basic”

  This appendix compares SBL with Microsoft Visual Basic.

- Appendix B—“Inscribe Tutorial”

  This appendix provides a tutorial for using the SBL Developer Kit.

- Appendix C—“Errors”

  This appendix provides information about Inscribe error codes.

- Appendix D—“Calling Executable Programs from Inscribe”

  This appendix provides information for Windows NT developers about how to directly call executable programs using the external procedures feature in Scalable SQL 4.

This manual also includes a glossary and an index.

Conventions

Unless otherwise noted, command syntax, code, and code examples use the following conventions:

Case	Commands and reserved words typically appear in uppercase letters. Unless the manual states otherwise, you can enter these items using uppercase, lowercase, or both. For example, you can type MYPROG, myprog, or MYprog.
[ ]	Square brackets enclose optional information, as in [*log_name*]. If information is not enclosed in square brackets, it is required.
|	A vertical bar indicates a choice of information to enter, as in [*file name* | @*file name*].
< >	Angle brackets enclose multiple choices for a required item, as in `/D=<5|6|7>`.
variable	Words appearing in italics are variables that you must replace with appropriate values, as in *file name*.
…	An ellipsis following information indicates you can repeat the information more than one time, as in [*parameter* …].
::=	The symbol ::= means one item is defined in terms of another. For example, a::=b means the item *a* is defined in terms of *b*.

chapter 1 Introduction

Inscribe is a development technology that helps you create and run scripts for use with Scalable SQL’s external procedures feature. External procedures allow you to access applications external to a database while you perform internal database operations. Inscribe uses Softbridge Basic Language (SBL), a programming language compatible with Visual Basic, to create scripts.

This chapter contains the following sections:

Features

Inscribe offers the following features:

- Allows you to distribute application logic by supporting client/server technology. Using external procedures written in Inscribe, you can divide application logic between client and server systems, allowing data processing to execute on the same system as the database.
- Allows you to access external applications (such as email and spreadsheets) and external devices (such as printers). You can also write scripts to query a Scalable SQL database and convert the data into an HTML document.
- Allows you to move compiled scripts that you write without platform-dependent code (such as message and dialog boxes) to all Scalable SQL platforms with little or no modification. (See Table 1-1 on page 12.)
- Allows you to write scripts that use an ODBC interface. This compatibility with ODBC enables you to access a wide variety of databases.

Note

Any scripts that you write for an ODBC interface support only the Microsoft platforms: Windows v3.x, Windows NT, and Windows 95.

- Provides a graphical interface for compiling, editing, and debugging SBL scripts with the SBL Development Tools.

Table 1-1 lists the platforms supported for each functional group. In addition, the following functional groups are supported on all platforms: Arrays, Compiler Directives, Control Flow, Dates and Times, Declarations, Errors, Files, Math Functions, Strings, Variants.

Table 1-1 Functional Platform Support

Functional Group	Windows v3.x	Windows 95	Windows NT	NetWare
Dialog Boxes	◆	◆	◆	
Dynamic Data Exchange (DDE)	◆	◆	◆	
Environmental Control	◆	◆	◆	❍
Objects	◆	◆	◆	
ODBC	◆	◆	◆	
Screen Input/Output	◆	◆	◆	⌘

❍ Date statement, Command, and Randomize only

⌘ Beep, Input function/statement, and Print only

Components

Inscribe has two pieces bundled with Scalable SQL, as follows:

- The Inscribe engine is bundled with the Scalable SQL engine. The Inscribe engine validates the arguments passed to Inscribe procedures. It is multi-threaded and supports multiple concurrent external procedure calls. It runs on the same machine where Scalable SQL runs, tightly coupling your application logic to your database.

  The Inscribe engine includes an SBL Interpreter, which loads and executes Inscribe procedures. Like the engine, the SBL Interpreter is multi-threaded and runs on the same machine where Scalable SQL runs. It provides an optimized environment for executing Inscribe procedures and returning output values to the engine.

- The Inscribe Developer Kit is bundled with the Scalable SQL Programming Interfaces. It contains the SBL Development Tools, which include an interactive utility that lets you edit, compile, and debug SBL scripts for bundling with your applications. Its features include Console and Variables windows for tracing code, code animation, breakpoints, and a dialog resource editor.

  The Inscribe Developer Kit also contains a set of tutorial files that help you create an SBL program that you can execute from a Scalable SQL trigger.

System Requirements

The Inscribe engine runs on Windows NT and NetWare operating environments.

The SBL Development Tools run on Windows v3.x, Windows NT, and Windows 95 operating environments.

Windows 3.x and Windows 95 are development environments only. This limitation means that you can only execute or test functionality in these environments that is not dependent on the presence of the Scalable SQL engine.

All Inscribe engine platforms can execute the applications you compile with the Development Tools as long as the code does not contain platform-dependent code, such as dialog boxes. For example, if you want to run scripts with identical behavior on Windows NT and NetWare, then you must include only functions that are supported on both platforms. See Table 1-1 on page 12 for more information about platform support.

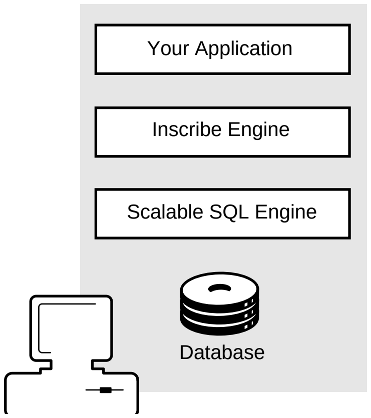
Inscribe Environment Configurations

When you use Inscribe to execute scripts associated with a database, the Inscribe engine must run on the same machine that runs the Scalable SQL engine. In addition, the compiled modules must be located in the same directory where the .DDF files for the database are located. (That is, the .DDF files do not necessarily have to be on the same machine where Scalable SQL is running.) Given these requirements, the following examples illustrate the possible system configurations.

Local Workstation Configuration

The local workstation configuration provides stand-alone operation. All components reside locally, and data files are stored on the workstation's disk drive.

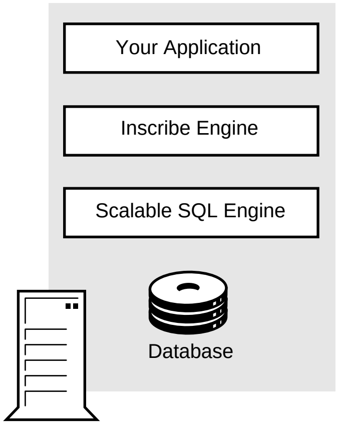
Figure 1-1 Local Workstation Configuration



Server-Based Configuration

In the server-based configuration, all components reside on the server, and data files are stored on the server. Alternatively, the data files could reside on another server.

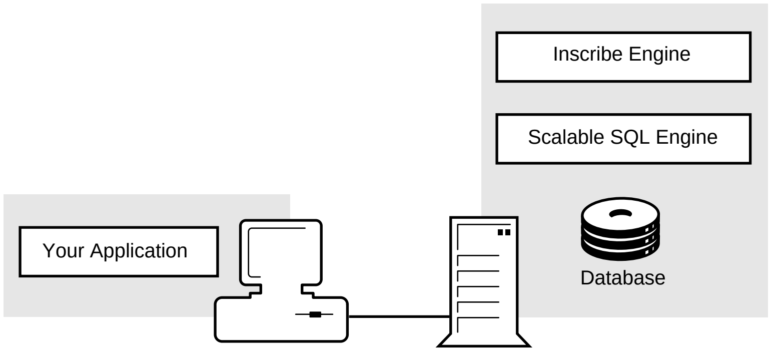
Figure 1-2 Server-Based Configuration



Client/Server Configuration

You can set up a client/server configuration in which the Inscribe and Scalable SQL engines run on a server and your application runs on a client workstation. In this client/server environment, your client application can execute Inscribe scripts on the server, but not on the client.

Figure 1-3 Client/Server Configuration



Development Procedure

In general, to develop Inscribe scripts with Scalable SQL, follow these steps:

1. Using the SBL Development Tools, create a Visual Basic compatible script and compile it to create a module.
2. Copy the compiled module to the directory that contains the Scalable SQL dictionary (.DDF) files.
3. Using Scalable SQL, issue CREATE PROCEDURE...EXTERNAL statements to define external procedure references for the module.

   This information is stored in the X$Proc system table (PROC.DDF file). Refer to the *SQL Language Reference* for more information about this system table and CREATE PROCEDURE syntax.
4. Invoke your script from the database using any of the following:
   - Direct Scalable SQL CALL statements
   - Internal stored procedures that execute CALL statements
   - Scalable SQL triggers that execute CALL statements

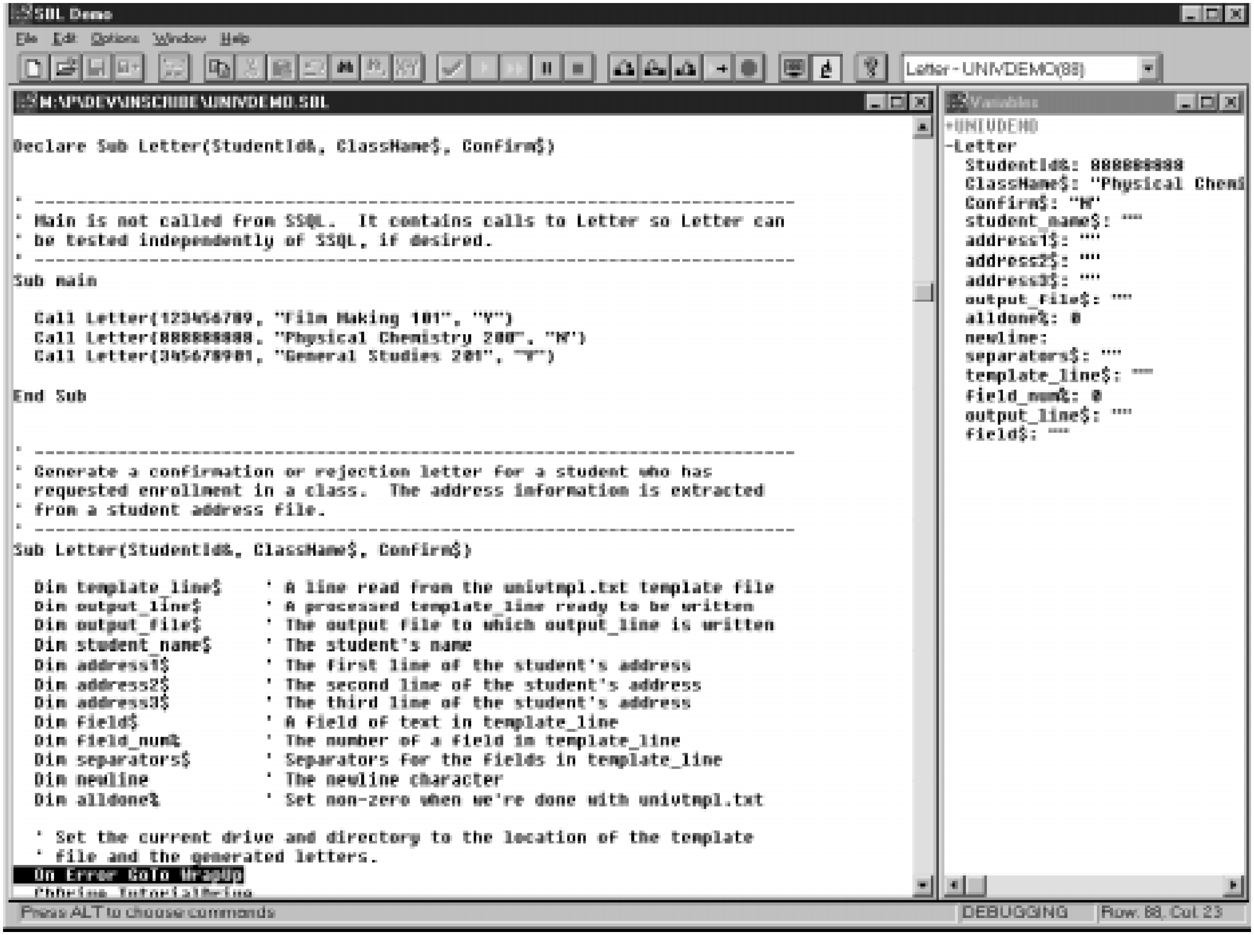
   Scalable SQL checks the procedure arguments against the procedure prototype and invokes the Inscribe engine to process the call. The Inscribe engine executes the procedure with the SBL Interpreter and returns any output arguments to the caller.

   Scalable SQL passes on any errors the Inscribe engine returns and updates the output arguments if no errors are returned. For more information about Inscribe status codes, refer to the *Status Codes and Messages* manual.

SBL Development Tools

SBLDemo, one of the SBL Development Tools, allows you to edit, compile, and debug SBL scripts. Figure 1-4 shows the main screen. For more information about using the Inscribe Developer Kit, refer to Appendix B, “Inscribe Tutorial” and the online help.

Figure 1-4 SBLDemo



```
Declare Sub Letter(StudentId&, ClassName$, Confirm$)


' ---------------------------------------------------------------------------
' Main is not called from SSQL.  It contains calls to Letter so Letter can
' be tested independently of SSQL, if desired.
' ---------------------------------------------------------------------------
Sub main

  Call Letter(123456789, "Film Making 101", "Y")
  Call Letter(888888888, "Physical Chemistry 200", "N")
  Call Letter(345678901, "General Studies 201", "Y")

End Sub


' ---------------------------------------------------------------------------
' Generate a confirmation or rejection letter for a student who has
' requested enrollment in a class.  The address information is extracted
' from a student address file.
' ---------------------------------------------------------------------------
Sub Letter(StudentId&, ClassName$, Confirm$)

  Dim template_line$     ' A line read from the univtmpl.txt template file
  Dim output_line$       ' A processed template_line ready to be written
  Dim output_file$       ' The output file to which output_line is written
  Dim student_name$      ' The student's name
  Dim address1$          ' The first line of the student's address
  Dim address2$          ' The second line of the student's address
  Dim address3$          ' The third line of the student's address
  Dim field$             ' A field of text in template_line
  Dim field_num%         ' The number of a field in template_line
  Dim separators$        ' Separators for the fields in template_line
  Dim newline            ' The newline character
  Dim alldone%           ' Set non-zero when we're done with univtmpl.txt

  ' Set the current drive and directory to the location of the template
  ' file and the generated letters.
  On Error GoTo WrapUp
```

```
+UNIVDEMO
-Letter
  StudentId&: 888888888
  ClassName$: "Physical Chemi
  Confirm$: "N"
  student_name$: ""
  address1$: ""
  address2$: ""
  address3$: ""
  output_file$: ""
  alldone%: 0
  newline:
  separators$: ""
  template_line$: ""
  field_num%: 0
  output_line$: ""
  field$: ""
```

```
Press ALT to choose commands          DEBUGGING     Row: 88, Col: 23
```

chapter 2 Language Overview

This chapter describes the essential rules and components of the Softbridge Basic Language (SBL). It contains the following sections:

Data Types

SBL supports standard Basic numeric, string, record, and array data. SBL also supports Dialog Records and Objects that the application defines.

You can declare data types for variables implicitly or explicitly, as follows:

- Implicitly on first reference by using a type character.
- Implicitly on first reference by omitting the type character, in which case the default type of VARIANT is assumed.
- Explicitly by using the Dim statement. You must explicitly declare variables of a user-defined type.

In any case, the variable can contain data of the declared type only.

Data Type Conversions

Basic performs automatic data conversions in the following cases:

- Between any two numeric types

  When converting from a larger type to a smaller type (for example, LONG to INTEGER), a runtime numeric overflow may occur. Such an error indicates that the number of the larger type is too large for the target data type. Loss of precision is not a run-time error (for example, when converting from double to single, or from either FLOAT type to either INTEGER type).

- Between fixed strings and dynamic strings

  When converting a fixed string to dynamic, Basic creates a dynamic string that has the same length and contents as the fixed string. When converting from a dynamic string to a fixed string, Basic may make some adjustments. If the dynamic string is shorter than the fixed string, the resulting fixed string is extended with spaces. If the dynamic string is longer than the fixed string, the resulting fixed string is a truncated version of the dynamic string. String conversions do not cause run-time errors.

- Between VARIANT and any other data type

  When required, Basic converts VARIANT strings to numbers. A type mismatch error occurs if the VARIANT string does not contain a valid representation of the required number.

No other implicit conversions are supported. In particular, Basic does not automatically convert between numeric and string data. Use the functions Val and Str**$** for such conversions. For more information about these and other functions, refer to the SBL online help.

VARIANT Data Type

You can use the VARIANT data type to define variables that contain any type of data. You store a tag with the VARIANT data to identify the type of data that it currently contains. You can examine the tag using the VarType function. The following table describes the tags and their meanings.

Tag	Name	Size of Data	Range
0	(Empty)	0	N/A

Tag	**Name**	**Size of Data**	**Range**
1	Null	0	N/A
2	Integer	2 bytes (short)	-32768 to 32767
3	Long	4 bytes (long)	-2.147E9 to 2.147E9
4	Single	4 bytes (float)	-3.402E38 to -1.401E-45 (negative) 1.401E-45 to 3.402E38 (positive)
5	Double	8 bytes (double)	-1.797E308 to -4.94E-324 (negative) 4.94E-324 to 1.797E308 (positive)
6	Currency	8 bytes (fixed)	-9.223E14 to 9.223E14
7	Date	8 bytes (double)	Jan 1st, 100 to Dec 31st, 9999
8	String	0 to 32 KB	0 to 32,767 characters
9	Object	N/A	N/A

Any newly-defined VARIANT is Empty by default, which signifies that the variable contains no initialized data. An Empty VARIANT converts to zero when used in a numeric expression and to an empty string in a string expression. You can test whether a VARIANT is uninitialized (that is, Empty) with the IsEmpty function.

Null VARIANTs have no associated data and serve only to represent invalid or ambiguous results. You can test whether a VARIANT contains a null value with the IsNull function. Null is not the same as Empty, which indicates that a VARIANT has not yet been initialized.

Arrays

You create arrays by specifying one or more subscripts at declaration or ReDim time. Subscripts specify the beginning and ending index for each dimension. If you specify an ending index only, the beginning index depends on the Option Base setting. You reference array elements by enclosing the appropriate number of index values in parentheses after the array name, as follows:

```
arrayname(a,b,c)
```

Dynamic arrays differ from fixed arrays in that you do not specify a subscript range for the array elements when you specify the array’s dimension. Instead, you set the subscript range using the Redim statement. With dynamic arrays, you can set the size of the array elements based on other conditions in your procedure. For example, you might want to use an array to store a set of values the user enters, but you do not know in advance how many values the user has. In this case, you dimension the array without specifying a subscript range and then execute a ReDim statement each time the user enters a new value. Alternatively, you can prompt for the number of values a user has and execute one ReDim statement to set the size of the array before prompting for the values.

If you use ReDim to change the size of an array and you want to preserve the contents of the array at the same time, be sure to include the Preserve argument to the ReDim statement. If you use Dim on a dynamic array before using the array, the maximum number of dimensions the array can have is 8. To create dynamic arrays with more dimensions (up to 60), do not use Dim on the array at all; instead, use only the ReDim statement inside your procedure.

For more information about the Dim and ReDim statements, refer to the SBL online help.

The following procedure uses a dynamic array, *varray*, to hold cash flow values the user enters:

```
Sub main
   Dim aprate as Single
   Dim cflowper as Integer
   Dim msgtext
   Dim x as Integer
   Dim netpv as Double
   cflowper=InputBox("Enter number of cash flow periods")
   For x= 1 to cflowper
     varray(x)=InputBox("Enter cash flow amount for &
        period #" & x & ":")
   Next x
   aprate=InputBox("Enter discount rate: ")

   If aprate>1 then
       aprate=aprate/100
   End If
   netpv=NPV(aprate,varray())
   msgtext="The net present value is: "
   msgtext=msgtext & Format(netpv, "Currency")
   MsgBox msgtext
  End Sub
```

Numbers

Numeric values are always signed. The following table shows the valid ranges of values.

Type	**From**	**To**
Integer	-32,768	32,767
Long	-2,147,483,648	2,147,483,647
Single	-3.402823e+38 0.0, 1.401298e-45	-1.401298e-45, (negative) 3.402823466e+38 (positive)
Double	-1.797693134862315d+308 0.0, 4.94065645841247d-308	-4.94065645841247d-308, (negative) 1.797693134862315d+308 (positive)
Currency	-922,337,203,685,477.5808	922,337,203,685,477.5807

Basic has no true Boolean variables. Basic considers 0 to be FALSE and any other numeric value to be TRUE. You can only use numeric values as Booleans. Comparison operator expressions always return 0 for FALSE and -1 for TRUE.

You can express integer constants in decimal, octal, or hexadecimal notation. You express decimal constants by using the decimal representation. To represent an octal value, precede the constant with *&O* or *&o* (for example, &O177). To represent a hexadecimal value, precede the constant with *&H* or *&h* (for example, &H8001).

Records

A record, or record variable, is a data structure that contains one or more elements, each of which has a value. Before declaring a record variable, you must define a type. Once you define the type, you can declare the variable to be of that type. The variable name should not have a type character suffix. You reference record elements using dot notation, as follows:

```
varname.elementname
```

Records can contain elements that are themselves records.

Dialog records look like any other user-defined data type. You reference elements using the *recname.elementname* syntax, where *recname* is the previously defined record name and *elementname* is a member of that record. The difference is that each element is tied to an element of a dialog. The application defines some dialogs; the user defines others.

Strings

Basic strings can be either fixed or dynamic. In either case, strings can vary in length from 0 to 32,767 characters. For fixed strings, you specify a length when you define the string, and you cannot change the length. Also, you cannot define a fixed string of zero length. Dynamic strings have no specified length.

There are no restrictions on the characters you can include in a string. For example, you can embed in strings the character whose ASCII value is 0 (NULL).

Arguments

You list arguments after the subroutine or function to which they apply. Whether you enclose the arguments in parentheses depends on how you want to pass the argument to the subroutine or function: either by value or by reference.

If you pass an argument by value, the variable used for that argument retains its value when the subroutine or function returns to the caller. If you pass an argument by reference, the variable's value is likely to change for the calling procedure. For example, suppose you set the value of variable *x* to 5 and pass *x* as an argument to a subroutine, named Mysub. If you pass *x* by value to Mysub, the value of *x* is always 5 after Mysub returns. However, if you pass *x* by reference to Mysub, *x* could be 5 or any other value resulting from the actions of Mysub.

To pass an argument by value, use one of the following syntax options:

```
CALL Mysub((x))
 Mysub(x)
 y=myfunction((x))
 CALL myfunction((x))
```

To pass an argument by reference, use one of the following options:

```
CALL Mysub(x)
 Mysub x
 y=myfunction(x)
 CALL myfunction(x)
```

You can declare external subroutines and functions (such as DLL functions) to accept arguments by value. In this case, those arguments are always passed by value.

When you call a subroutine or function that takes arguments, you usually supply values for those arguments by listing them in the order shown in the syntax for the statement or function. For example, suppose you define a function as follows:

```
Myfunction(id, action, value)
```

This syntax shows that the function called Myfunction requires three arguments: *id*, *action*, and *value*. When you call this function, you supply those arguments in the order shown. If the function contains just a few arguments, it is fairly easy to remember the order of each of the arguments. However, if a function has several arguments and you want to be sure the values you supply are assigned to the correct arguments, use named arguments.

Named arguments are identified by name rather than by position in the syntax. To use a named argument, use the following syntax:

```
namedarg := value
```

Using this syntax for Myfunction results in the following:

```
Myfunction id:=1, action:="get", value:=0
```

The advantage of named arguments is that you do not need to remember the original order in which they are listed in the syntax, so the following function call is also correct:

```
Myfunction action:="get", value:=0, id:=1
```

Named arguments have another advantage when calling functions or subroutines that have a mix of required and optional arguments. Ordinarily, you need to use commas as placeholders in the syntax for the optional arguments that you do not use. With named arguments, however, you can specify just the arguments you want to use and their values and forget about their order in the syntax. For example, Myfunction is defined as follows:

```
Myfunction(id, action, value, counter)
```

In this syntax, you can use named arguments in either of the following forms:

```
Myfunction id:="1", action:="get", value:="0"

Myfunction value:="0", counter:="10", action:="get", id:="1"
```

Note

Although you can shift the order of named arguments, you cannot omit required arguments. All SBL functions and statements accept named arguments. In the SBL online help, the argument names are listed in the syntax for each statement and function.

Dialog Boxes

To create and run a dialog, complete these steps:

1. Define a dialog record using the Begin Dialog...End Dialog statements and the dialog definition statements such as TextBox, OKButton.
2. *Optional*: Create a function to handle dialog interactions using the Dialog Functions and Statements.
3. Display the dialog using either the Dialog Function or Dialog Statement.

The following example code illustrates these steps.

Step 1:
Define the dialog box

```
Declare Function myfunc(identifier$, action, suppvalue)
Sub Main
  Begin Dialog NEWDLG dimx, dimy, caption, .myfunc
    ListBox....
    ComboBox....
    OKButton....
    CancelButton....
  End Dialog
  Dim dlg as NEWDLG
  Dim response as Integer
  response=Dialog(dlg)
  If response=-1 then
    'clicked OK button
  ElseIf response=0 then
    'clicked Cancel button
  ElseIf response>0 then
    'clicked another command button
  End If
End Sub
Function myfunc(identifier$, action, suppvalue)
  '...code to handle dialog box actions
End Function
```

Step 3:
Display the dialog box

Step 2:
Write a function to handle dialog box interaction

Defining a Dialog Box

The Begin Dialog...End Dialog statements define a dialog. The last parameter to the Begin Dialog statement is the name of a function, prefixed by a period (.). This function handles interactions between the dialog and the user.

The Begin Dialog statement supplies three parameters to your function: an identifier (a dialog control ID), the action taken on the control, and a value with additional action information. Your function should have these three arguments as input parameters. For more information about the Begin Dialog...End Dialog statements, refer to the SBL online help.

Writing a Dialog Function

You can write a function that defines dialog behavior. For example, your function could disable a check box, based on a user's action. The body of the function uses the *Dlg*-prefixed SBL statements and functions to define dialog actions.

Define the function itself using the Function...End Function statement or declare it using the Declare statement before using the Begin Dialog statement. Enter the name of the function as the last argument to Begin Dialog. The function receives three parameters from Begin Dialog and returns a value. Return any value greater than zero to leave the dialog open after the user clicks a command button (such as Help).

Displaying the Dialog Box

Use the Dialog function (or statement) to display a dialog. The argument to Dialog is a variable name that you previously dimensioned as a dialog record. The name of the dialog record comes from the Begin Dialog... End Dialog statement. The return values for the Dialog function determine which key was pressed:

- -1 for OK
- 0 for Cancel
- >0 for a command button

If the user clicks Cancel, the Dialog statement returns an error, which you can trap with the On Error statement.

Dialog Box Functions and Statements

The function you create uses the Dlg dialog functions and statements to manipulate the active dialog. This is the *only* function that can use these functions and statements. Following are the Dlg functions and statements:

Function or Statement	**Description**
DlgControlId	Returns numeric ID of a dialog control.
DlgEnable Function	Identifies whether a control is enabled or disabled.
DlgEnable Statement	Enables or disables a dialog control.
DlgFocus Function	Returns ID of the dialog control having input focus.
DlgFocus Statement	Sets focus to a dialog control.
DlgListBoxArray Function	Returns contents of a list box or combo box.
DlgListBoxArray Statement	Sets contents of a list box or combo box.

Function or Statement	Description
DlgText Function	Returns the text associated with a dialog control.
DlgText Statement	Sets the text associated with a dialog control.
DlgValue Function	Returns the value associated with a dialog control.
DlgValue Statement	Sets the value associated with a dialog control.
DlgVisible Function	Identifies whether a control is visible or disabled.
DlgVisible Statement	Shows or hides a dialog control.

Most of these functions and statements take a control ID as their first argument. For example, a check box is defined with the following statement:

```
CheckBox 20, 30, 50, 15, "My check box", .Check1
```

With this statement, `DlgEnable "Check1", 1` enables the check box, and `DlgValue("Check1")` returns 1 if the check box is currently checked, 0 if not. The IDs are case-sensitive and do not include the dot that appears before the ID's definition. Dialog functions and statements can also work with numeric IDs. Numeric IDs depend on the order in which you define the dialog controls.

For example, if the check box was the first control defined in the dialog record, then `DlgValue(0)` is equivalent to `DlgValue("Check1")`. (The control numbering begins from 0, and the Caption control does not count.) You find the numeric ID using the DlgControlID function.

For some controls (such as buttons and text) the last argument in the control definition, ID, is optional. If it is not specified, the text of the control becomes its ID. For example, you can refer to the Cancel button as *Cancel* if its ID is not specified in the CancelButton statement.

Dynamic Data Exchange

Dynamic Data Exchange (DDE) is a process by which two applications communicate and exchange data. One application can be your Basic program. To communicate with another application, you must open a connection, called a DDE channel, using the statement DDEInitiate. The application must already be running before you can open a DDE channel. To start an application, use the Shell command.

DDEInitiate requires two arguments: the DDE application name and a topic name. The DDE application name is usually the name of the .EXE file you use to start the application, without the .EXE extension. For example, the DDE name for Microsoft Word is WINWORD. The topic name is usually a file name with which to retrieve or send data, although there are some reserved DDE topic names, such as System. Refer to the application's documentation for a list of available topic names.

After you have opened a channel to an application, you can obtain text and numbers (using DDERequest), send text and numbers (using DDEPoke), or send commands (using DDEExecute). Because you have a limited number of channels available at once (depending on the operating system in use and the amount of memory you have available), you should close the DDE channel (using DDETerminate) when you have finished communicating with the application.

The other DDE command available in SBL is DDEAppReturnCode, which you use for error checking purposes. After retrieving or sending text or executing a command, you can use DDEAppReturnCode to make sure the application performed the task as expected. If an error occurred, your program can notify the user of the error.

Object Handling

Objects are the end products of a software application, such as a spreadsheet, graph, or document. Each software application has its own set of properties and methods that change the characteristics of an object.

Properties affect how an object behaves. For example, width is a property of a range of cells in a spreadsheet, colors are a property of graphs, and margins are a property of word processing documents.

Methods cause the application to do something to an object. Examples are Calculate for a spreadsheet, Snap to Grid for a graph, and AutoSave for a document.

In SBL, you have the ability to access an object and use the originating software application to change properties and methods of that object. Before you can use an object in a procedure, however, you must access the software application associated with the object by assigning it to an object variable. Then you attach an object name (with or without properties and methods) to the variable to manipulate the object. The syntax for doing this is shown in the following code example.

Figure 2-1 Handling Objects

Step 1:
Create an object variable to access the application

Step 2:
Use methods and properties to act on the objects

```
Sub main
 Dim visio as Object
 Dim doc as Object
 Dim page as Object
 Dim i as Integer, doccount as Integer
 Set visio = GetObject(,"visio.application")
  If (visio Is Nothing) then
      Set visio = CreateObject("visio.application")
      If (visio Is Nothing) then
         MsgBox "Couldn't find visio!"
         Exit Sub
      End If
 doccount = visio.documents.count
 For i = 1 to doccount
    Set doc = visio.documents(i)
    If doc.name = "myfile.vsd" then
       Set page = doc.pages(1)
       Exit Sub
    End If
 Next i
 Set doc=visio.documents.open("myfile.vsd")
 Set page = doc.pages(1)
End Sub
```

Note

The examples shown here are specific to the VISIO software application. Object, property, and method names vary from one application to another. For more information about the applicable names to use, refer to the software documentation for the application you want to access.

Creating an Object Variable

In Figure 2-1, the Dim statement creates an object variable called *visio* and assigns the application, VISIO, to it. The SET statement assigns the VISIO application to the variable *visio* using either GetObject or CreateObject. Use GetObject if the application is already open on the Windows desktop. Use CreateObject if the application is not open.

Using Methods and Properties

To access an object, property, or method, use the following syntax:

```
appvariable.object.property
```

```
appvariable.object.method
```

In Figure 2-1, `visio.documents.count` is a value returned by the Count method of the Document object for the VISIO application, which is assigned to the INTEGER variable `doccount`. Alternatively, you can create a second object variable and assign the Document object to it using VISIO's Document method, as the SET statement shows.

Expressions

An expression is a collection of two or more terms that perform a mathematical or logical operation. The terms are usually either variables or functions that are combined with an operator to evaluate to a string or numeric result. You use expressions to perform calculations, manipulate variables, or concatenate strings.

Expressions are evaluated according to precedence order. Use parentheses to override the default precedence order. Following is the precedence order (from high to low) for the operators:

- Numeric operators
- String operators
- Comparison operators
- Logical operators

Numeric Operators

Following are the numeric operators, presented in high-low precedence order:

Table 2-1 Numeric Operators

Operator	Description
^	Exponentiation.
-,+	Unary minus and plus.
*, /	Numeric multiplication or division. For division, the result is a DOUBLE.

Table 2-1 Numeric Operators *continued*

Operator	Description
\	Integer division. The operands can be INTEGER or LONG.
Mod	Modulus or Remainder. The operands can be INTEGER or LONG.
-, +	Numeric addition and subtraction. You can also use the + operator for string concatenation.

String Operators

Following are the string operators:

Table 2-2 String Operators

Operator	Description
&	String concatenation
+	String concatenation

Comparison Operators

Following are the comparison operators; these operators can operate on numeric and string data.

Table 2-3 Comparison Operators

Operator	Description
>	Greater than

Table 2-3 Comparison Operators *continued*

Operator	Description
<	Less than
=	Equal to
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal to

For numbers, the operands are widened to the least common type. The preferred order is as follows:

1. INTEGER
2. LONG
3. SINGLE
4. DOUBLE

For strings, the comparison is case-sensitive and based on the collating sequence that the user-specified language uses. The result is 0 for FALSE and -1 for TRUE.

Logical Operators

Following are the logical operators:

Table 2-4 Logical Operators

Operator	Description
Not	Unary Not. Operand can be INTEGER or LONG. The operation is performed bitwise (one’s complement).
And	And. Operands can be INTEGER or LONG. The operation is performed bitwise.
Or	Inclusive Or. Operands can be INTEGER or LONG. The operation is performed bitwise.
Xor	Exclusive Or. Operands can be INTEGER or LONG. The operation is performed bitwise.
Eqv	Equivalence. Operands can be INTEGER or LONG. The operation is performed bitwise. (A Eqv B) is the same as (Not (A Xor B)).
Imp	Implication. Operands can be INTEGER or LONG. The operation is performed bitwise. (A Imp B) is the same as ((Not A) OR B).

Error Handling

SBL contains three error handling statements and functions for trapping errors in your program: Err, Error, and On Error. SBL returns a code for many of the possible run-time errors you might encounter. Refer to Appendix C, "Errors" for a complete list of codes.

In addition to the errors that SBL traps, you can create your own set of codes for trapping errors specific to your program. For example, if your program establishes rules for file input and the user does not follow the rules, you can create specific errors to indicate these violations. You can trigger an error and respond appropriately using the same statements and functions you would use for SBL-returned error codes.

Regardless of the error trapped, you can handle errors in one of two ways:

- ◆ Place error-handling code directly before a line of code where an error might occur (such as a File Open statement).
- ◆ Label a separate section of the procedure just for error handling and force a jump to that label if any error occurs.

The On Error statement handles both options.

Trapping Errors SBL Returns

The following example shows the two ways to trap errors. Option 1 places error-handling code directly before the line of code that could cause an error. Option 2 contains a labeled section of code that handles any error.

Figure 2-2   Trapping Errors

```
Sub main
     Dim userdrive,userdir,msgtext
in1: userdrive=InputBox("Enter drive:",,"C:")
```

Option 1:
Place error-
handling code
within the body
of a procedure

```
     On Error Resume Next
     Err=0
     ChDrive userdrive
     If Err=68 then
        MsgBox "Invalid Drive. Try Again."
        Goto in1
     End If
```

Option 2:
Place error-
handling code
at the end of
a procedure
and Goto it via
a label

```
     On Error Goto Errhdlr1
in2: userdir=InputBox("Enter directory:")
     ChDir userdrive & "\" & userdir
     MsgBox "New default directory is:" & userdrive & "\" userdir
     Exit Sub
Errhdlr1:
     Select Case Err
      Case 75
       msgtext="Path is invalid."
      Case 76
       msgtext="Path not found."
      Case Else
       msgtext="Error" & Err & ":" & Error$ & "occurred."
     End Select
     MsgBox msgtext & "Try again."
     Resume in2
End Sub
```

Trapping Errors Within Code

The On Error statement identifies the line of code to go to in case of an error. In Option 1 of this example, the Resume Next parameter indicates that execution continues with the next line of code after the error, and the line of code to handle errors is the If statement. It uses the Err statement to determine which error code is returned.

Trapping Errors Using an Error Handler

The On Error statement used in Option 2 of Figure 2-2 specifies a label to jump to in case of errors. The code segment is part of the main procedure and uses the Err statement to determine which error code is returned. To make sure your code does not accidentally fall through to the error handler, precede it with an Exit statement.

Trapping User-Defined Errors

Figure 2-3 and Figure 2-4 show the two ways to set and trap user-defined errors. Both options use the Error statement to set the user-defined error to the value 30,000. To trap the error, option 1 places error-handling code directly before the line of code that could cause an error. Option 2 contains a labeled section of code that handles any user-defined errors.

Figure 2-3 Trapping User-Defined Errors: Option 1

Option 1: Place error-handling code within the body of a procedure

```
Sub main
     Dim custname as String
     On Error Resume Next
in1: Err=0
     custname=InputBox$("Enter customer name:")
     If custname="" then
       Error 30000
       Select Case Err
         Case 30000
           MsgBox "You must enter a customer name."
           Goto in1
         Case Else
           MsgBox "Undetermined error. Try again."
           Goto in1
       End Select
     End If
     MsgBox "The name is: " & custname
End Sub
```

Figure 2-4 Trapping User-Defined Errors: Option 2

Option 2: Place error-handling code at the end of a procedure and Goto it via a label

```
Sub main
     Dim custname as String
     On Error Goto Errhandler
in1: Err=0
     custname=InputBox$("Enter customer name:")
     If custname="" then
       Error 30000
     End If
     MsgBox "The name is: " & custname
     Exit Sub
Errhandler:
     Select Case Err
       Case 30000
         MsgBox "You must enter a customer name."
       Case Else
         MsgBox "Undetermined error. Try again."
     End Select
     Resume in1
End Sub
```

Derived Trigonometric Functions

You can write several trigonometric functions in Basic using the built-in functions. The following table lists several of these functions:

Function	**Computed By:**
Secant	Sec(x) = 1/Cos(x)
CoSecant	CoSec(x) = 1/Sin(x)
CoTangent	CoTan(x) = 1/Tan(x)
ArcSine	ArcSin(x) = Atn(x/Sqr(-x*x+1))
ArcCosine	ArcCos(x) = Atn(-x/Sqr(-x*x+1))+1.5708
ArcSecant	ArcSec(x) = Atn(x/Sqr(x*x-1))+Sgn(x-1)*1.5708
ArcCoSecant	ArcCoSec(x) = Atn(x/Sqr(x*x-1))+(Sgn(x)-1)*1.5708
ArcCoTangent	ArcTan(x) = Atn(x)+1.5708
Hyperbolic Sine	HSin(x) = (Exp(x)-Exp(-x))/2
Hyperbolic Cosine	HCos(x) = (Exp(x)+Exp(-x))/2
Hyperbolic Tangent	HTan(x) = (Exp(x)-Exp(-x))/(Exp(x)+Exp(-x))
Hyperbolic Secant	HSec(x) = 2/(Exp(x)+Exp(-x))
Hyperbolic CoSecant	HCoSec(x) = 2/(Exp(x)-Exp(-x))
Hyperbolic Cotangent	HCotan(x) = (Exp(x)+Exp(-x))/(Exp(x)-Exp(-x))

Function	**Computed By:**
Hyperbolic ArcSine	HArcSin(x) = Log(x+Sqr(x*x+1))
Hyperbolic ArcCosine	HArcCos(x) = Log(x+Sqr(x*x-1))
Hyperbolic ArcTangent	HArcTan(x) = Log((1+x)/(1-x))/2
Hyperbolic ArcSecant	HArcSec(x) = Log((Sqr(-x*x+1)+1)/x)
Hyperbolic ArcCoSecant	HArcCoSec(x) = Log((Sgn(x)*Sqr(x*x+1)+1)/x)
Hyperbolic ArcCoTangent	HArcCoTan(x) = Log((x+1)/(x-1))/2

chapter 3 Statements and Functions

The following table summarizes the Softbridge Basic Language (SBL) statements and functions by group. For detailed descriptions of each of these statements and functions, refer to the SBL online help.

Functional Group	**Function**	**Description**
Arrays	Erase	Reinitializes contents of an array.
	LBound	Returns the lower bound of an array's dimension.
	ReDim	Declares dynamic arrays and reallocates memory.
	UBound	Returns the upper bound of an array's dimension.
Compiler Directives	$CStrings	Treats a backslash in a string as an escape character, as in the C programming language.
	$Include	Tells the compiler to include statements from another file.
	$NoCStrings	Tells the compiler to treat a backslash as a normal character.
	Line Continuation	Continues a long statement across multiple lines.

Functional Group	Function	Description
Compiler Directives *continued*	Rem	Treats the remainder of the line as a comment.
Dates and Times	Date Function	Returns the current date.
	Date Statement	Sets the system date.
	DateSerial	Returns the date value for year, month, and day specified.
	DateValue	Returns the date value for the specified string.
	Day	Returns the day of month component of a date-time value.
	Hour	Returns the hour of day component of a date-time value.
	IsDate	Determines whether a value is a legal date.
	Minute	Returns the minute component of a date-time value.
	Month	Returns the month component of a date-time value.
	Now	Returns the current date and time.
	Second	Returns the second component of a date-time value.
	Time Function	Returns the current time.
	Time Statement	Sets the current time.

Functional Group	**Function**	**Description**
Dates and Times *continued*	Timer	Returns the number of seconds since midnight.
	TimeSerial	Returns the time value for hour, minute, and second specified.
	TimeValue	Returns the time value for the specified string.
	Weekday	Returns the day of the week for the specified date-time value.
	Year	Returns the year component of a date-time value.
Declarations	Const	Declares a symbolic constant.
	Declare	Forwardly declares a procedure in the same module or in a dynamic link library.
	Def*type*	Declares the default data type for variables.
	Dim	Declares variables.
	Function...End Function	Defines a function.
	Global	Declares a global variable.
	Option Base	Declares the default lower bound for array dimensions.
	Option Compare	Declares the default case sensitivity for string comparisons.

Functional Group	**Function**	**Description**
Declarations *continued*	Option Explicit	Forces all variables to be explicitly declared.
	ReDim	Declares dynamic arrays and reallocates memory.
	Static	Defines a static variable or subprogram.
	Sub...End Sub	Defines a subprogram.
	Type	Declares a user-defined data type.
Dialog boxes	Begin Dialog	Begins a dialog definition.
	Button	Defines a button dialog control.
	ButtonGroup	Begins definition of a group of button dialog controls.
	CancelButton	Defines a Cancel button dialog control.
	Caption	Defines the title of a dialog.
	CheckBox	Defines a check box dialog control.
	ComboBox	Defines a combo box dialog control.
	Dialog Function	Displays a dialog and returns the button pressed.
	Dialog Statement	Displays a dialog.
	DlgControlId	Returns the numeric ID of a dialog control.

Functional Group	**Function**	**Description**
Dialog boxes *continued*	DlgEnable Function	Identifies whether a dialog control is enabled or disabled.
	DlgEnable Statement	Enables or disables a dialog control.
	DlgEnd	Closes the active dialog.
	DlgFocus Function	Returns the ID of the dialog control having input focus.
	DlgFocus Statement	Sets focus to a dialog control.
	DlgListBoxArray Function	Returns contents of a list box or combo box.
	DlgListBoxArray Statement	Sets contents of a list box or combo box.
	DlgSetPicture	Changes the picture in the picture control.
	DlgText Function	Returns the text associated with a dialog control.
	DlgText Statement	Sets the text associated with a dialog control.
	DlgValue Function	Returns the value associated with a dialog control.
	DlgValue Statement	Sets the value associated with a dialog control.
	DlgVisible Function	Identifies whether a control is visible or hidden.
	DlgVisible Statement	Shows or hides a dialog control.
	DropComboBox	Defines a drop-down combo box dialog control.

Functional Group	**Function**	**Description**
Dialog boxes *continued*	DropListBox	Defines a drop-down list box dialog control.
	GroupBox	Defines a group box in a dialog.
	ListBox	Defines a list box dialog control.
	OKButton	Defines an OK button dialog control.
	OptionButton	Defines an option button dialog control.
	OptionGroup	Begins definition of a group of option button dialog controls.
	Picture	Defines a picture control.
	PushButton	Defines a push button dialog control.
	StaticComboBox	Defines a static combo box dialog control.
	Text	Defines a line of text in a dialog.
	TextBox	Defines a text box in a dialog.

Functional Group	**Function**	**Description**
Dynamic Data Exchange	DDEAppReturnCode	Returns a code from an application on a DDE channel.
	DDEExecute	Sends commands to an application on a DDE channel.
	DDEInitiate	Opens a DDE channel.
	DDEPoke	Sends data to an application on a DDE channel.
	DDERequest	Returns data from an application on a DDE channel.
	DDETerminate	Closes a DDE channel.
Environment Control	AppActivate	Activates another application.
	Command	Returns the command line specified when the MAIN subprogram ran.
	Date Statement	Sets the current date.
	DoEvents	Lets the operating system process messages.
	Environ	Returns a string from the operating system’s environment.
	Randomize	Initializes the random-number generator.
	SendKeys	Sends keystrokes to another application.
	Shell	Runs an executable program.

Functional Group	Function	Description
Errors	Assert	Triggers an error if a condition is false.
	Erl	Returns the line number at which a run-time error occurred.
	Err Function	Returns a run-time error code.
	Err Statement	Sets the run-time error code.
	Error	Generates an error condition.
	Error Function	Returns a string representing an error.
	On Error	Controls run-time error handling.
	Resume	Ends an error-handling routine.
	Trappable Errors	Defines errors that SBL code can trap.
Files	ChDir	Changes the default directory for a drive.
	ChDrive	Changes the default drive.
	Close	Closes a file.
	CurDir	Returns the current directory for a drive.
	Dir	Returns a file name that matches a pattern.
	Eof	Checks for end of file.
	FileAttr	Returns information about an open file.
	FileCopy	Copies a file.

Functional Group	**Function**	**Description**
Files *continued*	FileDateTime	Returns modification date and time of a specified file.
	FileLen	Returns the length of a specified file in bytes.
	FreeFile	Returns the next unused file number.
	Get	Reads bytes from a file.
	GetAttr	Returns the attributes of a specified file.
	Kill	Deletes files from a disk.
	Input, InputB Functions	Returns a string of characters (or bytes, for InputB) from a file.
	Input Statement	Reads data from a file or from the keyboard.
	Line Input	Reads a line from a sequential file.
	Loc	Returns the current position of an open file.
	Lock	Controls access to some or all of an open file by other processes.
	Lof	Returns the length of an open file.
	MkDir	Makes a directory on a disk.
	Name	Renames a disk file.
	Open	Opens a disk file or device for I/O.
	Print	Prints data to a file or to the screen.

Functional Group	**Function**	**Description**
Files *continued*	Put	Writes data to an open file.
	Reset	Closes all open disk files.
	RmDir	Removes a directory from a disk.
	Seek Function	Returns the current position for a file.
	Seek Statement	Sets the current position for a file.
	SetAttr	Sets the attribute information for a file.
	Spc	Displays the specified number of spaces.
	Tab	Moves the print position to the specified column.
	Unlock	Controls access to some or all of an open file by other processes.
	Width	Sets output-line width for an open file.
	Write	Writes data to a sequential file.
Flow control	Call	Transfers control to a subprogram.
	Do...Loop	Controls repetitive actions.
	Exit	Causes the current procedure or loop structure to return.
	For...Next	Loops a fixed number of times.
	Goto	Sends control to a line label.
	If...Then...Else	Branches on a conditional value.

Functional Group	**Function**	**Description**
Flow control *continued*	Let	Assigns a value to a variable.
	Lset	Left-aligns one string or a user-defined variable within another.
	On...Goto	Branches to one of several labels depending on value.
	Rset	Right-aligns one string within another.
	Select Case	Executes one of a series of statement blocks.
	Set	Sets an object variable to a value.
	Stop	Stops program execution.
	While...Wend	Controls repetitive actions.
	With	Executes a series of statements on a specified variable.
Math Functions	Abs	Returns the absolute value of a number.
	Atn	Returns the arc tangent of a number.
	Cos	Returns the cosine of an angle.
	Derived Functions	Computes other numeric and trigonometric functions.
	Exp	Returns the value of *e* raised to a power.
	Fix	Returns the integer part of a number.

Functional Group	Function	Description
Math functions *continued*	FV	Returns the future value of a cash flow stream.
	Int	Returns the integer part of a number.
	IPmt	Returns the interest payment for a specified period.
	IRR	Returns the internal rate of return for a cash flow stream.
	IsNumeric	Determines whether a value is a legal number.
	Log	Returns the natural logarithm of a value.
	NPV	Returns the net present value of a cash flow stream.
	Pmt	Returns a constant payment per period for an annuity.
	PPmt	Returns the principal payment for a specified period.
	PV	Returns the present value of a future stream of cash flows.
	Rate	Returns the interest rate per period.
	Rnd	Returns a random number.
	Sin	Returns the sine of an angle.
	Sgn	Returns a value indicating the sign of a number.

Functional Group	**Function**	**Description**
Math functions *continued*	Sqr	Returns the square root of a number.
	Tan	Returns the tangent of an angle.
Objects	Class List	Lists available classes.
	Clipboard	Accesses the Windows Clipboard.
	CreateObject	Creates an OLE2 automation object.
	GetObject	Retrieves an OLE2 object from a file or retrieves the active OLE2 object for an OLE2 class.
	Is	Determines whether two object variables refer to the same object.
	Me	Obtains the current object.
	New	Allocates and initializes a new OLE2 object.
	Nothing	Sets an object variable to not refer to an object.
	Object	Declares an OLE2 automation object.
	Typeof	Checks the class of an object.
	With	Executes statements on an object or a user-defined type.

Functional Group	Function	Description
ODBC	SQLClose	Closes a data source connection.
	SQLError	Returns a detailed error message for ODBC functions.
	SQLExecQuery	Executes a SQL statement.
	SQLGetSchema	Obtains information about data sources, databases, terminology, users, owners, tables, and columns.
	SQLOpen	Establishes a connection to a data source for other functions to use.
	SQLRequest	Makes a connection to a data source, executes a SQL statement, and returns the results.
	SQLRetrieve	Returns the results of a SELECT statement that the SQLExecQuery function executed into a user-provided array.
	SQLRetrieveToFile	Returns the results of a SELECT statement that the SQLExecQuery function executed into a user-specified file.
Screen Input/ Output	Beep	Produces a short beeping tone through the speaker.
	Input Function	Returns a string of characters from a file.
	Input Statement	Reads data from a file or from the keyboard.
	InputBox	Displays a dialog box that prompts for input.

Functional Group	**Function**	**Description**
Screen Input/ Output *continued*	MsgBox Function	Displays a Windows message box.
	MsgBox Statement	Displays a Windows message box.
	PasswordBox	Displays a dialog that prompts for input. Does not echo input.
	Print	Prints data to a file or to the screen.
Strings	Asc	Returns an integer corresponding to a character code.
	CCur	Converts a value to currency.
	CDbl	Converts a value to a double-precision floating point.
	Chr, ChrB	Converts a character (or byte, for ChrB) code to a string.
	CInt	Converts a value to an integer by rounding.
	CLng	Converts a value to a long by rounding.
	CSng	Converts a value to a single-precision floating point.
	CStr	Converts a value to a string.
	CVar	Converts a number or string to a variant.
	CVDate	Converts a value to a variant date.
	Format	Converts a value to a string using a picture format.

Functional Group	**Function**	**Description**
Strings *continued*	GetField	Returns a substring from a delimited source string.
	Hex	Returns the hexadecimal representation of a number as a string.
	InStr, InStrB	Returns the character (or byte, for InStrB) position of one string within another.
	LCase	Converts a string to lowercase.
	Left, LeftB	Returns the left portion of a string.
	Len, LenB	Returns the length of a string or size of a variable.
	Like Operator	Compares a string against a pattern.
	LTrim	Removes leading spaces from a string.
	Mid, MidB Function	Returns a portion of a string.
	Mid Statement	Replaces a portion of a string with another string.
	Oct	Returns the octal representation of a number as a string.
	Right, RightB	Returns the right portion of a string.
	RTrim	Removes trailing spaces from a string.
	SetField	Replaces a substring within a delimited target string.

Functional Group	**Function**	**Description**
Strings *continued*	Space	Returns a string of spaces.
	Str	Returns the string representation of a number.
	StrComp	Compares two strings.
	String	Returns a string consisting of a repeated character.
	Trim	Removes leading and trailing spaces from a string.
	UCase	Converts a string to uppercase.
	Val	Converts a string to a number.
Variants	IsEmpty	Determines whether a variant has been initialized.
	IsNull	Determines whether a variant contains a NULL value.
	Null	Returns a null variant.
	VarType	Returns the type of data stored in a variant.

appendix A SBL and Visual Basic

SBL shares a substantial common core of functions and statements with Microsoft's Visual Basic; however, each one has unique capabilities. SBL supports the following functional areas:

- Arrays
- Compiler directives
- Control flow
- Date and time functions
- Declarations
- Dynamic Data Exchange
- Environment control
- Error handling
- File control
- Math functions
- Object handling
- ODBC
- Screen I/O
- Variants

In addition, SBL offers the following statements and functions that are not available in Visual Basic:

- $CStrings
- $Include
- $NoCStrings
- Assert
- GetField$
- SetField$

Visual Basic does not have a syntax to create or run dialogs. In contrast, SBL has a set of functions and statements that enable using dialogs. (One modified version of Visual Basic, available in products such as Excel, is called Visual Basic for Applications, or VBA, and does provide dialog handling statements and functions.)

SBL does not support the following Visual Basic v3.0 and v4.0 constructs because they represent outdated syntax:

- Control arrays
- GoSub ... Return
- Resume (*lineNumber*)
- On GoSub
- Goto *lineNumber*

Table A-1 shows the Visual Basic v3.0 constructs that SBL does not support.

Table A-1 Visual Basic v3.0 Constructs Not Supported by SBL

Clipboard Functions	GetData SetData
Variant Support	IsError IsObject
Array Support	Array() For Each ... Next Arrays in records
Financial Functions	DDB MIRR NPer SLN SYD
Date Functions	DateDiff DateAdd DatePart
Control Flow	Iif Switch Choose End
OLE Support	OLE Server

Table A-1 Visual Basic v3.0 Constructs Not Supported by SBL *continued*

Other	Property Get, Property Set, and Property Let IMEStatus LoadPicture LoadResData, LoadResPicture, and LoadResString Partition QBColor RGB CVErr MS Jet database engine and all related functions VB visual objects and all related functions VB VBX custom controls VB non-visual objects (such as App and Debug) VB property procedures

Table A-2 shows the Visual Basic v4.0 constructs that SBL does not support.

Table A-2 Visual Basic v4.0 Constructs Not Supported by SBL

Directives	#Const #Else #ElseIf #EndIf #If
Data Types	Boolean data types Byte data types Collection data types Variant error subtype
Boolean Functions	DefBool CBool

Table A-2 Visual Basic v4.0 Constructs Not Supported by SBL *continued*

Byte Functions	CByte DefByte
Char Function	ChrW
Clipboard Functions	GetData SetData
MsgBox Function	System modal option
Variant Support	IsError IsObject IsArray CVErr VarType support for boolean, byte, and OLE Arrays in variants
Array Support	Array(), Control arrays and IsArray For Each ... Next Arrays in records SafeArrays in OLE calls
Financial Functions	DDB MIRR NPer SLN SYD
Date Functions	DefDate DateDiff DateAdd DatePart CDate

Table A-2 Visual Basic v4.0 Constructs Not Supported by SBL *continued*

VBA Registry	GetAllSettings GetSettings DeleteSetting SaveSetting
Control Flow	Iif Switch Choose GoSub ... Return Resume (*lineNumber*) On GoSub Goto *lineNumber* End
OLE Support	Error object Additional OLE parameters (such as LPSTR) OLE Server
New Parameters to Functions	FileAttr, `attribute=32` Format, new arguments FreeFile, takes an argument Weekday, new argument MsgBox, *helpfile* and *context* arguments InputBox, *helpfile* and *context* arguments Password Box, *helpfile* and *context* argument

Table A-2 Visual Basic v4.0 Constructs Not Supported by SBL *continued*

Other	Error Size is not a LONG Naked END statement doEvents as a function SPC and TAB in print Property Get, Property Set, and Property Let AscB IMEStatus LoadPicture LoadResData, LoadResPicture, and LoadResString Collection object Debug object only supports print Partition QBColor RGB CVErr ComInput, ComOutput StrConv DefObject Private / Public declarations TypeName MS Jet database engine and all related functions VB visual objects and all related functions Windows 95 custom controls VB VBX custom controls VB OCX custom controls VB non-visual objects (such as App and Debug) VB property procedures Remote OLE automation

appendix B Inscribe Tutorial

This appendix provides a tutorial that guides you through editing, compiling, debugging, and executing an SBL script. The tutorial contains the following sections:

- “Compiling an SBL Script”
- “Debugging a Compiled Module”
- “Executing a Compiled Module”

The sample SBL script (UNIVDEMO.SBL) used in this tutorial generates a confirmation or rejection letter for a student who is enrolling in a class. The subprogram Letter has three parameters: a student ID number, a class name, and an indicator that specifies whether to generate a confirmation letter or a rejection letter. A trigger in the Scalable SQL database uses this sample script to generate confirmation letters when students successfully enroll in a class and rejection letters when they cannot enroll because the class is full.

You can access the SBLDemo on-line help by selecting the Help button, as shown below.

Compiling an SBL Script

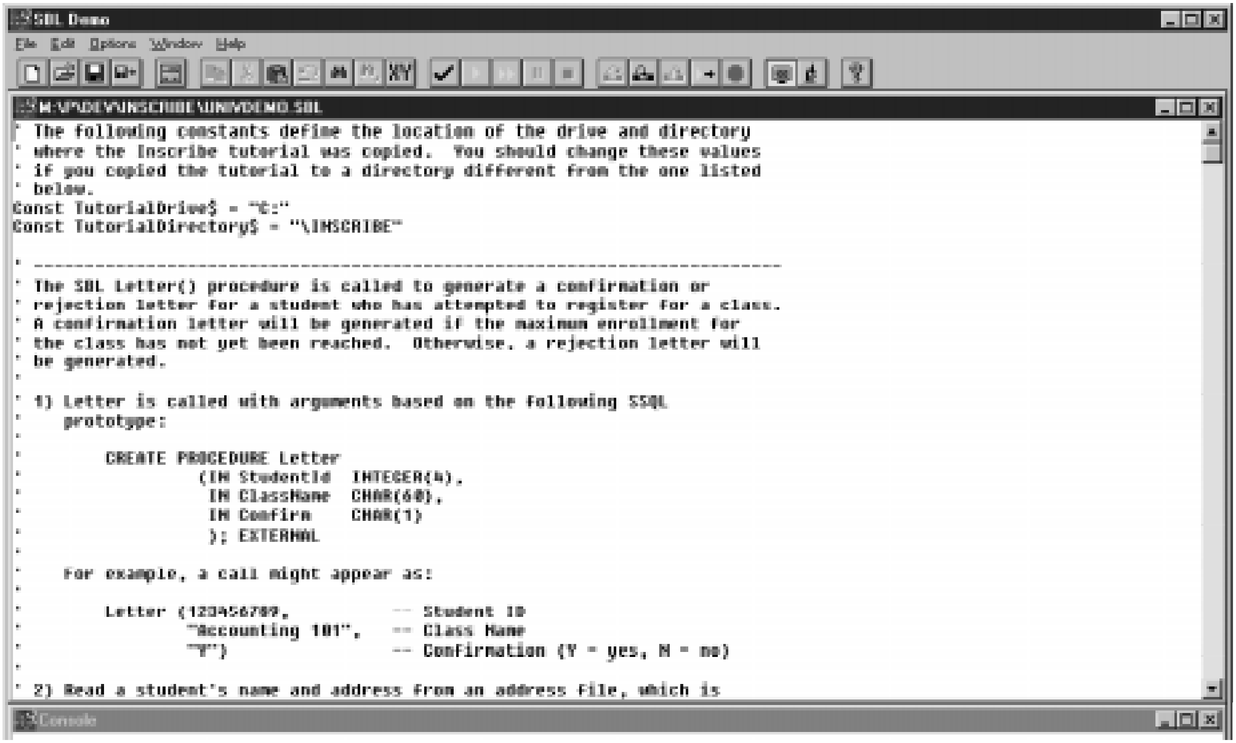
In this section, you open the sample SBL script and compile it.

1. Copy this tutorial from the INTF\INSCRIBE\TUTORIAL subdirectory in the Scalable SQL installation directory to a temporary directory on the same server where you have installed Scalable SQL and Inscribe.

   For example, if you are running Scalable SQL on a NetWare server and have drive K: mapped to a volume on that server, then you can create and use the temporary directory K:\INSCRIBE. If you are running Scalable SQL on a Windows NT workstation, you can create and use the temporary directory C:\INSCRIBE, where C: is a local drive on your Windows NT workstation.

2. Start the utility for developing SBL scripts by double-clicking the SBLDemo icon.

3. Use the Open command on the File menu to open the file UNIVDEMO.SBL, located in the temporary directory you created in Step 1. The following screen shows SBLDemo with the UNIVDEMO.SBL file open.



```
' The following constants define the location of the drive and directory
' where the Inscribe tutorial was copied.  You should change these values
' if you copied the tutorial to a directory different from the one listed
' below.
Const TutorialDrive$ = "C:"
Const TutorialDirectory$ = "\INSCRIBE"

' ------------------------------------------------------------------------
' The SBL Letter() procedure is called to generate a confirmation or
' rejection letter for a student who has attempted to register for a class.
' A confirmation letter will be generated if the maximum enrollment for
' the class has not yet been reached.  Otherwise, a rejection letter will
' be generated.
'
' 1) Letter is called with arguments based on the following SSQL
'    prototype:
'
'        CREATE PROCEDURE Letter
'                 (IN StudentId  INTEGER(4),
'                  IN ClassName  CHAR(60),
'                  IN Confirm    CHAR(1)
'                  ); EXTERNAL
'
'    For example, a call might appear as:
'
'        Letter (123456789,          -- Student ID
'                "Accounting 101",   -- Class Name
'                "Y")                -- Confirmation (Y = yes, N = no)
'
' 2) Read a student's name and address from an address file, which is
```

4. At the beginning of the script, update the TutorialDrive$ and TutorialDirectory$ values to the location where you copied the tutorial files, as in the following example:

```
Const TutorialDrive$ = "K:"

Const TutorialDirectory$ = "\INSCRIBE"
```

5. In the Main subprogram, type a few semicolons after one of the calls to the Letter procedure, as in the following example:

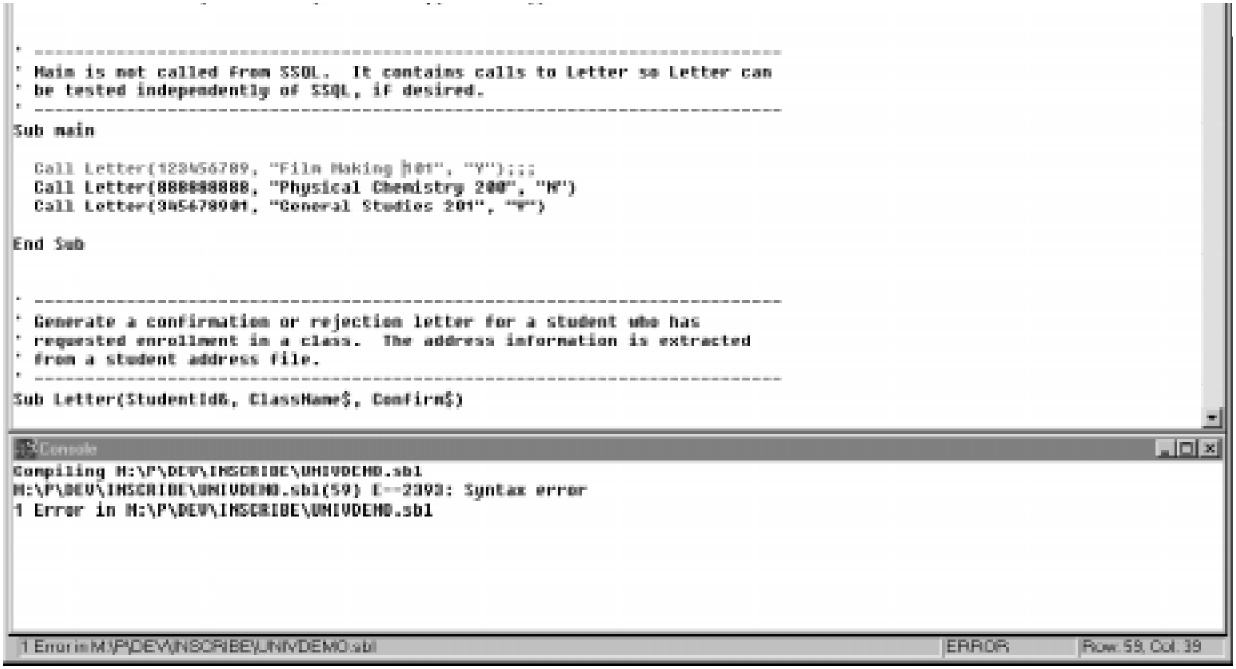
```
Call Letter (123456789, “Film Making 101”, “Y”);;;
```

These semicolons are syntactically incorrect and allow you to see what SBLDemo displays when it detects a compilation error.

6. Select the Console Window button (shown below) to display the Console window at the bottom of the SBLDemo screen.

7. Use the Compile command on the File menu to compile the script.

SBLDemo displays an error message in the Console window and highlights the line with the extra semicolons in red. The following example shows SBLDemo with an error:



```
' ---------------------------------------------------------------------------
' Main is not called from SSQL.  It contains calls to Letter so Letter can
' be tested independently of SSQL, if desired.
' ---------------------------------------------------------------------------
Sub main

  Call Letter(123456789, "Film Making 101", "Y");;;
  Call Letter(888888888, "Physical Chemistry 200", "N")
  Call Letter(345678901, "General Studies 201", "Y")

End Sub


' ---------------------------------------------------------------------------
' Generate a confirmation or rejection letter for a student who has
' requested enrollment in a class.  The address information is extracted
' from a student address file.
' ---------------------------------------------------------------------------
Sub Letter(StudentId&, ClassName$, Confirm$)
```

Console

```
Compiling H:\P\DEV\INSCRIBE\UNIVDEMO.sbl
H:\P\DEV\INSCRIBE\UNIVDEMO.sbl(59) E--2393: Syntax error
1 Error in H:\P\DEV\INSCRIBE\UNIVDEMO.sbl
```

```
1 Error in M:\P\DEV\INSCRIBE\UNIVDEMO.sbl          ERROR          Row: 59, Col: 39
```

8. Delete the extra semicolons and use the Compile command to compile the script again.

SBLDemo displays a message in the Console window indicating a successful compilation. This compilation generates a module (UNIVDEMO.SBX) from the source script (UNIVDEMO.SBL) and saves the module in the location you specified in Step 4.

Debugging a Compiled Module

In this section, you debug the sample module you compiled in the previous section. This section assumes that SBLDemo is running and the UNIVDEMO.SBL script is open and compiled.

1. In the Main subprogram, place the cursor on the line containing the second Call Letter statement and select the Toggle Breakpoint button (shown below) to set a breakpoint on that statement.

   SBLDemo highlights the breakpoint line in blue.

2. Execute the module up to the line containing the breakpoint. To execute the module, use either the Run command on the File menu or the Execute button (shown below).

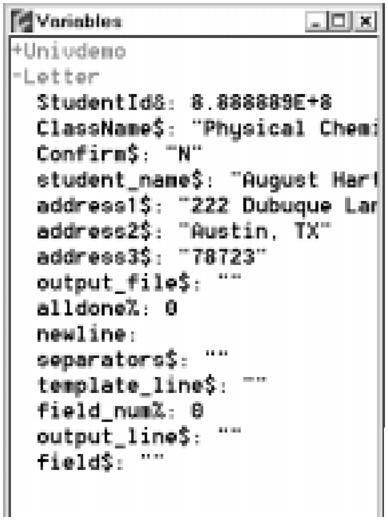
3. When the execution stops at the breakpoint, select the Watch Window button (shown below), to display the Variables window listing variables that are currently active.

   At this time, no variables are active since no variables are defined in the Main subprogram.

4. Select the Step Into button (shown below) to step into the Letter procedure.

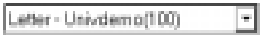
5. Select the Step Over button (shown below) several times to see the values of the variables change.

The following illustration shows sample contents of the Variables window.



```
Variables
+Univdemo
-Letter
  StudentId&: 8.888889E+8
  ClassName$: "Physical Chem
  Confirm$: "N"
  student_name$: "August Har
  address1$: "222 Dubuque Lar
  address2$: "Austin, TX"
  address3$: "78723"
  output_file$: ""
  alldone%: 0
  newline:
  separators$: ""
  template_line$: ""
  field_num%: 0
  output_line$: ""
  field$: ""
```

6. Select the Letter - Univdemo list box (shown below) to display the current call stack.



Letter - Univdemo(100)

Because the Main subprogram called the Letter procedure, two entries appear on the call stack—one for Letter and one for Main.

7. Select the Execute button.

The module generates letters using the file TEMPLATE.TXT as a template. For each call to the Letter procedure, the module matches the value of the student ID parameter to the student name and address in the ADDRESS.TXT file.

8. Examine the resulting three letters created in the tutorial directory:

   - JACKSON.TXT for student ID 123456789 (John Jackson)
   - HARTIG.TXT for student ID 888888888 (August Hartig)
   - SMITH.TXT for student ID 345678901 (John Smith)

Note

In this tutorial, all development occurs in the same directory that stores the database. However, you may choose to develop your SBL scripts in a location other than your database directory. If so, copy all the .SBX files to your database directory before you execute these scripts from a Scalable SQL application. This ensures that Inscribe can locate all the modules.

When Inscribe loads a module in a Scalable SQL database, all the public functions and procedures defined in that module, except for the Main subprogram, are automatically registered. Inscribe does not register Main subprograms for two reasons:

- Multiple modules can contain Main subprograms. If Inscribe did not ignore the Main subprogram definition, Inscribe would generate duplicate definition errors.
- Because Main subprograms are ignored, they provide a convenient place for inserting debugging code that tests the other functions and procedures in the module.

Executing a Compiled Module

In this section, you use SQLScope to execute a series of Scalable SQL statements in the file UNIVDEMO.SQL. These statements perform a series of operations on an empty database to accomplish the following:

- Create tables, procedures, and a trigger that you can use to enroll students in a class called General Studies and check that the enrollment of General Studies does not exceed its maximum size.
- Perform a series of enrollments into General Studies to demonstrate how acceptance and rejection letters are generated when a Scalable SQL trigger calls the Letter procedure in the UNIVDEMO.SBX module.
- Restore the database to its original state by dropping the tables, procedures, and triggers created.

Note

Before starting SQLScope, you may wish to print a copy of UNIVDEMO.SQL so you can more easily refer to the comments contained in that file.

Perform the following steps to execute the compiled tutorial module:

1. Using the Scalable SQL Setup utility, assign a name to the empty tutorial database. (Your named database can be bound or not bound.)
2. Start SQLScope and log in to the empty tutorial database using the name you assigned in Step 1.
3. Use the Environment command on the Settings menu to display the environment settings. Ensure that the Statement Separator is #.
4. Use the Open command on the File menu to open and load the file UNIVDEMO.SQL from the temporary directory where you copied the tutorial.

5. Select the First button in the Run window to execute the first statement in UNIVDEMO.SQL.

   SQLScope displays Status Code -101, indicating that the first statement, a SET statement, completed successfully.

6. Select the Next button to execute subsequent statements in UNIVDEMO.SQL.

   SQLScope displays status information for each statement. Continue selecting the Next button until all the statements are executed. (When all statements have been executed, the Next button is no longer available.)

   All the statements should complete successfully, except for the two Insert statements that attempt to enroll the students Jane Doe and John Smith. These two statements should return status 911 because the students could not be enrolled since the General Studies class is full.

7. Exit SQLScope.

8. In the tutorial directory, examine the three acceptance letters and the two rejection letters that the Letter procedure created. The three acceptance letters are as follows:

   HAPPY.TXT for student ID 777777777 (Anthony Happy)

   HARTIG.TXT for student ID 888888888 (August Hartig)

   JACKSON.TXT for student ID 123456789 (John Jackson)

   The two rejection letters are as follows:

   DOE.TXT for student ID 234567890 (Jane Doe)

   SMITH.TXT for student ID 345678901 (John Smith)

appendix C Errors

The following table lists the run-time errors that Softbridge Basic Language (SBL) returns. You can trap these errors using On Error. You can use the Err function to query the error code, and then use the Error function to query the error text.

Error Code	**Error Text**
5	Illegal function call
6	Overflow
7	Out of memory
9	Subscript out of range
10	Duplicate definition
11	Division by zero
13	Type mismatch
14	Out of string space
19	No resume
20	Resume without error
28	Out of stack space
35	Sub or function not defined

Error Code	**Error Text**
48	Error in loading DLL
52	Bad file name or number
53	File not found
54	Bad file mode
55	File already open
58	File already exists
61	Disk full
62	Input past end of file
63	Bad record number
64	Bad file name
68	Device unavailable
70	Permission denied
71	Disk not ready
74	Cannot rename with different drive
75	Path/file access error
76	Path not found
91	Object variable set to nothing

Error Code	Error Text
93	Invalid pattern
94	Illegal use of NULL
102	Command failed
429	Object creation failed
438	No such property or method
439	Argument type mismatch
440	Object error
901	Input buffer would be larger than 64K
902	Operating system error
903	External procedure not found
904	Global variable type mismatch
905	User-defined type mismatch
906	External procedure interface mismatch
907	Push button required
908	Module has no MAIN
910	Dialog box not declared

appendix D Calling Executable Programs from Inscribe

In addition to calling compiled Inscribe modules, Windows NT developers can directly call executable programs using the external procedures feature in Scalable SQL v4.0.

Note

NetWare Developers: You cannot run executable programs directly from Scalable SQL. This feature is supported in the Windows NT environment only.

Scalable SQL provides an interface to both executable programs and Inscribe modules through the Scalable SQL CREATE PROCEDURE and CALL statements. However, the arguments you pass to executable programs are different from the arguments you pass to Inscribe modules, as follows:

- The first argument must be an INTEGER(4) that is IN or INOUT. This argument is an input flag that tells Inscribe whether it should wait for the executable to complete. If you set this argument to 0, Inscribe waits for the executable to complete; if you set this argument to 1, Inscribe does not wait.
- The second argument must be an INTEGER(4) that is OUT or INOUT. Inscribe sets this output argument to the exit code returned by either the executable (if Inscribe waits for the executable to complete) or the child process ID (if Inscribe does not wait).
- Inscribe converts the remaining arguments to strings, if necessary, and passes them as input arguments on the executable's command line.

The following examples show how you can define and then call the executable XCOPY.EXE using Scalable SQL statements:

Example 1

```
CREATE PROCEDURE xcopy(IN  FlagsINTEGER(4),
     OUT StatusINTEGER(4),
     IN  SourcePathCHAR(64),
     IN  DestPathCHAR(64),
     IN  OptionsCHAR(20));
     EXTERNAL
```

Example 2

```
CALL xcopy(0, status, ‘C:\INS\DIR1’, ‘C:\INS\DIR2’, ‘/E /I’)
```

The first argument is the input flag, which tells Inscribe to wait for XCOPY to complete. Because the first argument is 0, Inscribe returns XCOPY’s exit code in the second argument. Inscribe passes the remaining arguments to XCOPY on the command line. Thus, given the previous examples, Inscribe creates and executes the following command line:

```
xcopy  C:\INS\DIR1  C:\INS\DIR2  /E /I
```

If the same external procedure name exists for both an executable and a Visual Basic procedure in an Inscribe module, Inscribe runs the Visual Basic procedure.

Glossary

call by reference

A procedure can modify arguments passed by reference to the procedure. Procedures written in Basic are defined by the language specification to receive their arguments by reference. If you call such a procedure and pass it a variable, and if the procedure modifies the formal parameter that corresponds to the variable, then the variable itself is also modified.

If you call such a procedure and pass it an expression, a temporary value is created for the expression and passed to the procedure. If the procedure then modifies the formal parameter that corresponds to the expression, the temporary value is also modified. However, the caller cannot access the temporary value, and the temporary value is discarded when the procedure returns.

call by value

When you pass an argument by value to a procedure, the called procedure receives a copy of the argument. If the called procedure modifies its corresponding formal parameter, it has no effect on the caller. Procedures written in other languages (such as C) can receive their arguments by value.

comment

A comment is text that documents a program. Comments have no effect on the program (except for metacommands). In Basic, a comment begins with a single quote and continues to the end of the line. If the first character in a comment is a dollar sign ($), the comment is interpreted as a metacommand. Lines beginning with the keyword *Rem* are also interpreted as comments.

control ID

A control ID can be either a numeric ID or a text string, in which case it is the name of the control. Control IDs are case-sensitive and do not include the dot that appears before the ID’s definition. Numeric IDs depend on the order in which you define the dialog controls. You can find the numeric ID using the DlgControlID function.

dialog control

A dialog control is an item in a dialog, such as a list box, combo box, or command button.

function

A function is a procedure that returns a value. In Basic, you specify the return value by assigning a value to the name of the function, as if the function were a variable.

label

A label identifies a position in the program at which to continue execution, usually as a result of executing a GoTo statement. To be recognized as a label, a name must begin in the first column, and must be immediately followed by a colon (":"). Reserved words are not valid labels.

metacommand

A metacommand is a command that instructs the compiler on how to build the program. In Basic, you specify metacommands in comments that begin with a dollar sign ($).

module

A module is a compiled script. All module file names have a .SBX extension.

name

A Basic name must start with a letter (A through Z). The remaining part of a name can also contain numeric digits (0 through 9) or an underscore character (_). A name cannot be more than 40 characters in length. Type characters are not considered part of a name.

precedence order

The precedence order is the system SBL uses to determine which operators in an expression to evaluate first, second, and so on. Operators with a higher precedence are evaluated before those with lower precedence. Operators with equal precedence are evaluated from left to right. The default precedence order (from high to low) is: numeric, string, comparison, logical.

procedure

A procedure is a series of SBL statements and functions that are executed as a unit. Both subprograms (Sub) and functions (Function) are procedures.

SBL

SBL is an acronym for the Softbridge Basic Language.

script

A script is a set of Visual Basic compatible procedures.

subprogram

A subprogram is a procedure that does not return a value.

type character

A type character is a special character you use as a suffix to the name of a function, variable, or constant. The character defines the data type of the variable or function. The characters are as follows:

Dynamic String	$
Integer	%
Long integer	&
Single precision floating point	!
Double precision floating point	#
Currency exact fixed point	@

vartype

A vartype is the internal tag that identifies the type of value currently assigned to a variant. Vartypes are as follows:

Empty	0	Double	5
Null	1	Currency	6
Integer	2	Date	7
Long	3	String	8
Single	4	Object	9

Index

User Comments

Pervasive Software would like to hear your comments and suggestions about our manuals. Please write your comments below and send them to us at:

Pervasive Software Inc.
Documentation
8834 Capital of Texas Highway
Austin, Texas 78759 USA

Inscribe User's Guide
100-003246-004
February 1998

Telephone: 1-800-287-4383
Fax: 512-794-1778
Email: docs@pervasive.com

Your name and title: ______________________________
Company: ______________________________
Address: ______________________________
____

Phone number: ________________ Fax: ________________

You may reproduce these comment pages as needed so that others can send in comments also.

I use this manual as: ❑ an overview ❑ a tutorial ❑ a reference ❑ a guide

	Excellent	Good	Fair	Poor
Completeness	❑	❑	❑	❑
Readability (style)	❑	❑	❑	❑
Organization/Format	❑	❑	❑	❑
Accuracy	❑	❑	❑	❑
Examples	❑	❑	❑	❑
Illustrations	❑	❑	❑	❑
Usefulness	❑	❑	❑	❑

Please explain any of your above ratings: ______________________________

____

____

____

In what ways can this manual be improved? ______________________________

____

____

____

____

____

You may reproduce these comment pages as needed so that others can send in comments also.