Superbase NG Programmer's Guide

Getting to Grips with the SIMPOL Language

Neil Robinson

Superbase NG Programmer's Guide: Getting to Grips with the SIMPOL Language

by Neil Robinson

Table of Contents

List of Tables

List of Examples

Chapter 1. Introduction

Copyright Information

Disclaimer

You are strongly recommended to take a backup of your system before a major installation and to make backups at regular intervals.

New Versions of this Document

Newer versions of this document will undoubtedly be released from time to time. It is recommended that you always ensure that you have the latest version of the documentation. Normally the latest version will be included with any update of the main product.

Software Used

This book was written using DocBook 5. It was initially written and edited in the Superbase NG IDE and eventually very late in production was switched to the <oXygen/> editor. A single source in XML is used to produce the book in HTML, HTML Help, and in PDF formats.

Part I. Quick Start With SIMPOL

This part of the book is intended to provide the reader with a quick introduction to the SIMPOL language without getting too bogged down in detail. It should, however, provide a useful and rapid introduction to the language for anyone who has experience in any other BASIC-oriented or C-oriented programming language.

Table of Contents

Chapter 2. Introduction

This book provides an introduction and reference guide to programming in SIMPOL, the new Superbase NG programming language for cross-platform development.

A program in SIMPOL is made up of functions. The normal entry point to a program is the function called `main()`. The most basic possible SIMPOL program is:

```
function main()
end function
```

Local Variables, Objects, and Values

Within a program there are local variables, objects and values. Local variables have to be defined as being of a particular type and can only be used for that type. Every object has a type, which is determined when the object is created. Values can be of any type that is permitted for a literal value, or may be null or infinite.

Consider the following program:

```
function main()
  string s
  s = "Hello world"
end function s
```

The first statement, **string s**, defines a local variable called `s`. The second statement, **s = "Hello world"**, assigns the value `"Hello world"` to the local variable `s`. However there is more to the assignment than it might at first appear. Local variables do not hold values, they refer to objects, and it is objects that contain values. In the above program the assignment to `s` of the string literal `"Hello world"` causes a new string object to be created, the string literal value is put in the new object and the local variable `s` is set to refer to that object.

Constants

There are three kinds of constant values in a SIMPOL program, string literals, numeric literals and intrinsic constants. String literals can be delimited by either single or double quotes, but the ending delimiter must match the starting delimiter. This allows single and double quotes to be used in strings without any special escape sequences, e.g.: `"Can't"` or `'Fred said "no"'`. Numeric literals can be specified in one of several bases by starting them with a leading zero and then a base indicator letter, e.g.: `0d100` (decimal), `0xff` (hexadecimal), `0o377` (octal) or `0b11001101` (binary). If no leading zero and base indicator are found then the number is assumed to be decimal. Intrinsic constants are identifiers that have a single fixed value. The permitted constants are `.nul` (the null value), `.inf` (infinity), `.true` (the boolean true value) and `.false` (the boolean false value).

A function can return a value by specifying that value after the **end function** that terminates the function, for example:

```
function main()
end function "The end of the program"
```

or

```
function main()
  string s
  s = "Hello world"
end function s
```

If no return value is specified then the return value is the null constant, `.nul`. A function can be called from within an expression, and may take parameters in the normal way, for example the following program will have a final return value of `15`:

```
function main()
  integer i
  i = 5
  i = i + double(i)
end function i

function double(integer i)
end function i+i
```

Function Parameters

Function parameters can be named and take a default value. Actual parameters supplied when the function is called can be specified by name, and if not specified at all will take the default value. When passing parameters to a function named parameters are resolved and passed first, and then unnamed actual parameters are passed to the unused function parameters from left to right. Finally any unpassed parameters are set to their default values, or if none is specified unused parameters are set to `.nul`.

For example, the following program produces a result value of: `x = abc, y = <y not specified>`

```
function main()
end function x_and_y(x = 'abc')

function x_and_y(string x = '<x not specified>', string y = \
                 '<y not specified>')
end function "x = " + x + ", y=" + y
```

Also, the next program produces a result value of: `x = xyz, y = <y not specified>, z = uvw`

```
function main()
end function x_and_y_and_z(z = 'uvw','xyz')

function x_and_y_and_z(string x = '<x not specified>', string y =\
  '<y not specified>', string z = '<z not specified>')
end function "x = " + x + ", y=" + y + ", z= " + z
```

Functions, types and local variables all have names, and parameters can be named. In order to be valid a name must start with a letter and all other characters in it must be a letter, a digit or an underscore.

Statements

Within a function SIMPOL is made up of statements. Typically a statement will occupy a single line within the source code, but more than one statement can be placed on one line by separating them with semi-colons (`;`) or colons (`:`). Also it is possible to break a statement into more than one line; the backslash (`\`) character is used to indicate that the following line is a continuation of the current one, but is only valid if it is the last non-white-space character on the line.

Intrinsic Functions

In order to manipulate values SIMPOL provides *intrinsic functions*. These are functions whose parameters are unnamed and must all be supplied. The return from an intrinsic function is some other value, which depends on, and only on, the parameter values. For example the **.len()** function takes a single string parameter and returns the number of characters it contains. As another example, the **.tostr()** function takes two numeric values, the first being a value to be represented as a string and the second being the base to use, e.g.: **.tostr(13,2)** returns '`1101`', or **.tostr(123456,10)** returns the string '`123456`'.

Operators

In addition to intrinsic functions there are operators which work on one or more parameters. Common examples of these are the addition operator `+`, the unary negation operator `-` which converts a value to its opposite, in some sense depending on the value type, and the comparison operators, `==`, `<` etc., which return a boolean value that depends on a comparison between the left and right operands. It should be noted that operators operate on values, not objects. For example the code:

```
integer i
integer j
i = 3
j = - i
```

does not change the value in the object referred to by `i` from `3`, it takes the value from that object, negates the value only, and assigns the result to `j`.

Complex Object Types

The object types that are also value types, such as string or boolean, exist primarily to hold values of that type. More complicated types exist either to provide information to the program, or to allow the program to do something. For example the `fsfileinputstream` type is used to read data from a file in a file system (such as on disk or over a network). With the simple value types an object is created for a local variable to refer to when an assignment is made to that local variable, as in the **string s;s = 'Hello world'** example earlier. With non-value types this is not the case — it is necessary to explicitly create these objects, normally using a `new()` function. The following example creates an input stream to read from the file `c:\autoexec.bat`:

```
function main()
  fsfileinputstream f

  f =@ fsfileinputstream.new("c:\autoexec.bat")
end function
```

The `=@` operator in this example is an important one which caused the local variable `f` to be set to refer to the object on the right hand side of the operator. This should be contrasted with the `=` operator which instructs the value on the right hand side to be assigned to the object referred to by the local variable on the left. As a rather pedantic yet instructive example, the following program has a resulting value of `1`:

```
function main()
  integer i
  integer j

  i = 1
  j = i
  i = 3
end function j
```

whereas the following program produces a result of `3`:

```
function main()
  integer i
  integer j

  i = 1
  j =@ i
  i = 3
end function j
```

The difference is that **j = i** assigns to the integer object referred to by `j` the value of the integer object referred to by `i`, whereas **j =@ i** causes both `i` and `j` to refer to the same integer object, so when a value is assigned to `i` it is setting the value of the object to which `j` refers.

The input stream is destroyed (closing the file) when the local variable `f` is destroyed, at the end of the function. In the case of an input stream, the parameters that have to be passed to a new function depend on the type of object being created, in this case only the filename is required.

Object types also have properties, which are either embedded objects of other types or references to other objects. For example the `fsfileinputstream` type has a property called `filename` of type string, which contains the name of the file being read. Extending the previous example slightly gives a program the result of which is the value '`c:\autoexec.bat`':

```
function main()
  fsfileinputstream f

  f =@ fsfileinputstream.new("c:\autoexec.bat")
end function f.filename
```

Object types can also have member functions, or methods, which are functions that do something with or to the object. For example the `fsfileinputstream` type has a member function called `getstring`, which can be used to read a string from the input file. The following program has a return value that is the first line of the `c:\autoexec.bat` file. The exact syntax for member functions such as `getstring` is type and function dependent, and can be found in the language reference.

```
function main()
  fsfileinputstream f

  f =@ fsfileinputstream.new("c:\autoexec.bat")
end function f.getstring(.inf,1,.true,.char(13)+.char(10))
```

Flow Control

There are flow control constructs in SIMPOL that are similar to many other languages. The **if … else if … else … end if** construct is straightforward:

```
if (i == 3)
  j = 5
else if (i == 4)
  j = 6
else
  j = 7
end if
```

In some cases the **.if()** intrinsic function provides a better alternative to the **if** construct. The **while … end while** can have conditional expressions at the beginning or the end, or both. The contained block will be entered if the initial expression is absent or is considered to have passed. At the end of the block control will return to the first test if the final condition is absent or is considered to have failed. The end while condition should therefore be thought of as an **end while if** type of expression. For example the following kind of loop could be used to process the lines in an `autoexec.bat` file:

```
fsfileinputstream f
string line
boolean error

f =@ fsfileinputstream.new("c:\autoexec.bat")
error = .false
while (f.endofdata == .false)
  line = f.getstring(.inf,1,.true,.char(13)+.char(10))
  // ...
  // process the line maybe setting error to true
  // if an error is encountered
  // ...
end while (error == .true)
```

File Types

As is the case with many programming language products, SIMPOL uses a number of different file types in the course of creating a program. This consists of source files, compiled modules, runnable programs, debug information and so on. In this section we will look at each of these areas in more detail.

Source Files

A SIMPOL program starts with one or more source files that are either in some ANSI 8-bit character encoding or else in Unicode (specifically in the pure 2-bytes per character form, not yet supporting windowing using aggregates). Unicode files should begin with a byte-order mark (BOM) in the form of Unicode character FEFF. If the data in the source file is in big endian (most significant byte first) order (typical for the Macintosh, Amiga, Atari ST, and other operating systems that are running or began on computers based on the Motorola CPU's starting with the MC68000) then the BOM must also be written in the big endian style, as FEFF. If the data is stored in little endian order, then the BOM must be written as FFFE. If the data is in Unicode and there is no BOM then the compiler will try to guess by analyzing the input. Files saved using the SIMPOL IDE will always have the correct BOM for Unicode files. Non-Unicode files do not have this issue, but have the problem that the 8-bit encoding of the source may not be interpreted in the same way on another computer or operating system if any characters above character value 127 are used.

Source files stored in ANSI format are typically stored with the extension `sma`. Those stored in Unicode format are stored with the extension `smu`. These extensions are not mandatory, but are recommended for interoperability with others and with any tools that may be created that may depend on the file name extensions. Source files stored by SIMPOL IDE's will always be stored in little-endian order even on big-endian operating systems.

Compiled Files

A compiled program in SIMPOL typically ends with the extension `smp`. Another type of compiled SIMPOL program is a module. Unlike a program, a module does not have a `main()` function. It is pre-compiled and can either be loaded at runtime using the system function `!loadmodulefile()` or it can be concatenated onto the end of a compiled SIMPOL program in a type of static linking. Module files are normally given the extension `sml`. The SIMPOL IDE is designed to allow for both static and dynamic linking (whereby the dynamic onking is done by the SIMPOL program when it needs to load a module, not by some loader). This allows the greatest amount of flexibility when designing complex programs using SIMPOL since modules can be written to be reused by other programs and therefore do not have to be compiled into every single program that uses them. It also allows components to be compiled only when their source files have changed and means that not every component must be compiled into one monolithic program. This will result in faster compile times.

One important issue that is related to storing reusable functionality in separate modules is that of scope and visibility. Unless expressly exported in the source code, user-defined types and functions are not visible outside of the module. To make them visible, add the `export` keyword to the end of the function or type declaration as shown below:

```
type mytype1(mytype) embed export
  string ID embed
  type(mytype) next
end type

type mytype2(mytype) embed export
  string ID embed
  type(mytype) next
  integer index embed
end type

function getid(type(mytype) t) export
end function t.ID
```

Debug Information

Debugging information comes in various flavors. In SIMPOL we have reserved the extension `smd` for files that store debugging information as part of the compilation process. These files are not yet in use at the time of writing, but will be used in the future. Another extension that *is* in use currently is `slg`. This is a file that is created when an error occurs while running a SIMPOL program using one of the debug versions of the loader, either the standard loader file or the CGI version. When an error occurs, the debug versions of these programs will output a file of the same name as the program that is executing, but with the file name extension `slg`. This file will contain the error message and error number together with the reconstructed source code of the function with an indicator as to which line contained the problem.

Part II. SIMPOL Language Basics

In this part of the book the basic language syntax of the SIMPOL language is covered in depth. This is not a book for teaching programming, however, so some experience in another language will be helpful in getting up to speed with this material quickly. This part is fairly dry and dusty but provides a well-rounded grounding in the essentials of the language itself. It does not cover the mechanics of compiling and running a program, that can be found elsewhere.

Table of Contents

Chapter 3. Basic SIMPOL Grammatical Stucture

Every programming language has various characteristics about the way that it is expected to be presented that can reasonably be termed its *grammar*. In this chapter we will discuss the points that will assist you in writing programs in SIMPOL. Although SIMPOL derives from BASIC in some respects, like any good language there are a number of elements that differentiate it from other languages that exist.

BASIC is a line-oriented language, whereas C and Pascal are statement-oriented. SIMPOL is mainly a statement-oriented language, but similarly to BASIC, it is not necessary to close a string at the end of a line, it will be done for you if you forget. More significant is that in SIMPOL every program begins at the `main()` function and ends when it returns from that function. There are no labels nor is there any equivalent of the BASIC keywords GOTO and GOSUB. For some programmers a more significant departure will be the total absence of global variables. In spite of, or perhaps because of these differences program design in SIMPOL is fast and effective and results in very fast and easily followed and maintained code.

End of Statement Characters

There are several ways to indicate the end of a statement in SIMPOL, using a semi-colon `;`, a colon `:`, and by starting a new line. These can also be mixed in a program, there is no requirement that they be used consistently.

Line Continuation Character

One common problem that occurs in languages like BASIC and SIMPOL that consider the end of line to also be the end of statement, is that it results in very long lines in the source programs. One method of combatting this problem is by use of a line continuation character. A line continuation character tells the parser or interpreter to continue reading the statement on the next line without ending the statement. In SIMPOL this character is the backslash (\) character. The line continuation character even applies within a string literal, as long as the backslash is the last character on the line other than white space before the end of line is reached. The only exception to this rule is in the case of the double slash comment, which means the entire line following is a comment.

Line Breaks and White Space

Line breaks are normally interpreted as end of statement characters unless they are suppressed by the programmer through the use of a line continuation character. White space characters (spaces, tabs, and end of line characters) are considered to be token separators similar to other punctuation such as parentheses and are otherwise ignored except when found within string literals.

Comments

Any string literal that is an L-value (found on the left side or beginning of a statement) is considered to be a comment. Such comments can even include the backslash character at the end of the line to allow them to extend across multiple lines. These comments are not considered to be line comments, but rather statement comments. If the string literal is closed then whatever follows it on the same line will not be considered a comment.

Another way of commenting is to use the double slash `//` to comment out an entire line. This comment type is line based and will also ignore the backslash line continuation character if it is at the end of the line. There is no method in the language for making block comments.

Literals

There are various types of literals in SIMPOL. Boolean literals are either `.true` or `.false`. Numeric literals must be a contiguous sequence of digits or digits and letters in the case of hexadecimal values. If only digits are encountered then the value is considered to be decimal. There is also support for other numeric bases. If it starts with:

- `0b` then the number will be evaluated as binary
- `0o` then the number will be evaluated as octal
- `0d` then the number will be evaluated as decimal
- `0x` then the number will be evaluated as hexadecimal

String literals can be delimited either by a single (`'`) or double (`"`) quote character. Any character can also be inserted into a string literal by placing the hexadecimal character value inside of curly braces. For example, to insert a carriage return and linefeed pair into a string literal it would look like this: `"{0D}{0A}"`. To escape the starting curly brace in the string simply include a second one. It is not always necessary to escape the starting curly brace, only if there is any chance that the following character could be interpreted as a hexadecimal value.

To include a double quote character in a string the easiest method is to use single quotes to delimit the string literal. Another method is simply to escape the double quote character with itself. An example of this is:

```
function main()
  string foo
  foo = 'Don''t you know?'
end function foo
```

which will return the string: `Don't you know?`.

Number literals are currently not supported, nor is scientific notation. To enter a value that has a fractional component into a number variable, it is necessary to place the value in a string and use the `.toval()` intrinsic function to convert it to a value. To enter a value that is normally a repeating decimal into a number, the special notation for repeating decimals can be used. This is also the way that repeating decimals are converted using the `.tostr()` intrinsic function. For example, to enter the value 1/3 into a number variable, try the following:

```
function main()
  number n
  n = .toval("0.3[3]", .nul, 10)
  n = n * 3
end function n
```

The return value of the function is `1`.

The special values `.inf` and `.nul` are essentially typeless, and can apply to any value type. The special value `.nul` can also mean the absence of an object.

Gotcha

These special values may cause some confusion when you begin using them. Any numeric expression, regardless of whether that is string concatenation or multiplication of integers, that includes the value `.nul` is equal to `.nul`. If you are adding strings together and one of them is equal to `.nul`, then the entire resulting string is equal to `.nul` and if that is being output, then nothing will be output. The same is true of `.inf` unless the expression also includes `.nul`.

Warning

Constants in SIMPOL can currently only be of type integer, string, or boolean and additionally integer constants cannot be negative (since this is considered to be an operation and at the time that constants are evaluated operations are not supported).

Case-sensitivity

All identifiers: type names, variable names, function names, and the names of symbolic constants are case-sensitive. All keywords, intrinsic, and system function names are also case-sensitive. This means that the variable names: foo, FOO, fOO, Foo, … are all considered to be different variables.

Identifiers

Identifiers in SIMPOL are currently composed of the characters a-z, A-Z, the digits 0-9, and the underscore. They are required to begin with an alphabetic character. The restriction to the character range may be modified in the future although we do not currently foresee supporting right-to-left identifier names or identifiers written using kanji characters.

Reserved Words

There has been a significant attempt made to keep the number of reserved words in SIMPOL to a minimum. The following words should be considered to be the current list of reserved words:

- and
- AND
- boolean
- constant
- else
- embed
- export
- end
- function
- if
- include

- information
- integer
- mod
- not
- number
- or
- OR
- reference
- resolve
- string
- type
- while
- XOR

This list should not yet be considered complete. Possible additions to the list could include:

- break
- case
- dim
- in
- par
- select
- switch
- sql
- where

Chapter 4. Data Types, Values, and Ranges

In this chapter we will discuss the various simple and complex data types that are present in SIMPOL. We will also discuss the valid value ranges, the special values `.nul` and `.inf`, functions, supplied complex types, and user-defined types.

To start with, it is important to point out that in SIMPOL, everything is an object. Types are objects, functions are objects, events are objects, and a variable always refers either to an object or to `.nul`. The scalar data types are also objects, but they are relatively simple objects. In this chapter we will cover the SIMPOL scalar data types: blob, boolean, integer, number, and string. We will also cover some standard included object types: array, date, datetime, and time. Other types will be discussed in depth in their own chapters, such as file streams, sockets, and databases.

Blobs

Blobs provide the SIMPOL programmer with a very powerful data type and mechanism for dealing with raw binary data. Blobs are also value types but they have a number of additional methods related to the ways in which they are most likely to be used.

It is probably easiest to think of blobs as being like a traditional array of bytes and at the same time to be very similar in the way they work to strings. Blobs can be concatenated using the `+` operator and then assigned to a variable of type blob. The new blob will be the combined length of the two original blobs. It is also possible to create a blob with a pre-determined size. This can be a significant performance improvement over strings, since it is possible to index into the blob using the square brackets operators `[]` and to then modify the value, whereas with a string concatenation would be used. When done once there is no significant difference but in a loop, concatenation would result in a large number of less efficient memory allocations that wouldn't have been necessary in a blob.

Booleans

Unlike many programming languages, SIMPOL includes a true Boolean data type and in what is becoming typically classic SIMPOL style, it can have four distinct values, `.true`, `.false`, `.nul`, and `.inf`. An important point to remember when using expressions in statements such as `while` and `if` is that unless the result of the expression is the boolean value `.true`, then it may very likely be considered to be false.

Integers

The SIMPOL integer data type is capable of containing the whole number values, both positive and negative plus zero, in an extremely large range. The maximum size of an integer is essentially unlimited, but is still somewhat operating system dependent. On virtually all operating systems it will support values up to 10 to the 150,000, and on 32-bit or greater operating systems will support values to 10 to the 4 billion or so. As can be seen, there is very little likelyhood that an integer will ever be larger than can be stored in SIMPOL. What *is* possible, however, is that an integer might be too large to be stored on a small device that has very little memory.

Numbers

The number data type in SIMPOL does not make use of floating point and is therefore not affected by the typical rounding errors that are found in floating point. The values stored in objects of type number are completely precise. Repeating decimal values are stored internally with complete accuracy. They can also be output in such a way as to indicate that they are repeating values and they can be converted back from strings to numbers with no loss of accuracy. An important point to remember when working with numbers in SIMPOL is that if you don't want values to have hundreds or even thousands of significant digits after the decimal point, then it is imperative that you make use of the `.fix()` intrinsic function to reduce the value to the precision and scale desired.

Note

Currently it is not possible to assign a decimal value to a number in the source code. To assign a decimal value, use a string and the `.toval()` function. Also, when a number is output to a string, if it contains a repeating decimal then that will be output in the following format: `<value>.<non-repeating portion><portion that repeats>[<portion that repeats>]` . For example, `3.3333333333333<repeating>` would be output as `3.3[3]`. The same value can be assigned from a string to a number using `.toval()`.

Strings

Strings in SIMPOL are fairly straightforward except for two significant issues: they are in Unicode format and they are essentially unlimited in size (limited only by memory). There are numerous intrinsic functions and types that are meant to work with strings and other than when leaving or entering SIMPOL those all work based on the character, not the byte. There are, however, methods that allow the user to specify one or two bytes per character when both reading and writing.

Pre-Defined Values

There are four pre-defined values in SIMPOL, two of which can be applied to to every type, and two that are specific to the boolean type. These values are: `.true`, `.false`, `.nul`, and `.inf`. The first two have been discussed in the section on the boolean type. The latter two can be applied to all value types and `.nul` can be applied to all types, both value and object types. The value `.inf` stands for infinity. It has many similarities to the `.nul` value and in some cases it is converted to that value if there is no available value to represent infinity, such as in SQL. The infinity that is represented in SIMPOL is both positive and negative infinity, but there is no value that represents infinitesimal (1 divided by infinity). In fact, in SIMPOL **1 / .inf** is equal to `.nul`.

It is important to understand how the two special values, `.nul` and `.inf` are used within SIMPOL and how their very existence in the language plays a role in how programs written in it may or may not work as expected. The first of the two values, `.nul` is used in many places as a return value and also it is the default value of a variable that has been created but not yet been initialized with a value. Having the concept of a null value in the programming language is quite useful, especially when interacting with databases where the desire to retain the characteristic of an empty field within a calculation may be desirable. The null value in SIMPOL follows some fairly clear rules. The value `.nul` combined with any other value or values results in the value `.nul`. If in your program you are suddenly finding an unexpected null result, then chances are that somewhere a value was uninitialized (or a database field is empty).

Functions

In SIMPOL functions are also a data type. It is perfectly reasonable to create a variable of type function that can then be used as a reference to a function. In fact, in some of the functionality provided with SIMPOL, such as tcpsocketserver it is necessary to provide a function reference so that the server knows which function to call when a socket connection is made.

Although that may seem to present considerable complexity, using function references is typically not necessary for programming with SIMPOL, other than when working with events, but the existence of this capability is one of the facilities that allows advanced programmers to create highly sophisticated programs. By using function references, it is possible to assign a display function to a function reference based on the type of the data that is to be displayed, and then just call the display function using the function reference. This makes for very clear and easy to read program code and moves the functionality of how to display a given data type out of the main program and into a function where it can also be reused.

Supplied Types

This section will be a continual work in progress, since the supplied types will be continually growing with time. There are two kinds of types: value types and object types. Value types are those previously listed and which are typically also known as scalar types in other languages. The following value types are included:

- blob
- boolean
- integer
- number
- string

These types have been discussed in earlier sections, so they won't be covered again here.

There are a number of different classes of object types. The list below is in no way exhaustive. See the SIMPOL Language Reference Manual for a full list of the available types.

- anyvalue
- array
- cgicall
- date
- datetime
- event
- fsfileinputstream
- fsfileoutputstream
- lock1
- ppcstype1
- ppcstype1field
- ppcstype1file
- ppcstype1index
- ppcstype1record
- ppcstype1server

- ppcstype1serverfield
- ppcstype1serversbme
- ppcstype1servertable
- ppcstype1serverudpport
- rgb
- sbme1
- sbme1field
- sbme1file
- sbme1index
- sbme1newfield
- sbme1newfile
- sbme1newindex
- sbme1record
- tcpsocket
- tcpsocketserver
- time
- UTOSdirectory
- UTOSdirectoryentry
- wxform
- wxformbutton
- wxformcheckbox
- wxformcombo
- wxformedittext
- wxformlist
- wxformoption
- wxformtext
- wxwindow

Object types are more complex than value types and normally must be initalized with the `new()` function and the return value must be assigned using the `=@` operator. The reason for this is efficiency. It would be terribly inefficient to completely initialize a large complex object every time a variable is created if the programmer only intends to use the variable to refer to an existing object. Think of a variable representing a window. That is quite a lot of processing and resource overhead if the window must be created as soon as the variable is created and then the window would be thrown away as soon as the variable is assigned to a different pre-existing window object. Also, some objects are created only by virtue of the existence of another object, so they cannot be created using a `new()` method. Such an object is the ppcstype1field, which can't exist without a ppcstype1file.

The `@=` operator and the `=@` operator are the equivalent of the **SET** command in SBL and the **Set** command in Visual Basic. It is also important to be able to test for the existence of an object. In SBL the `IS()` function combined with the `NOTHING` keyword is the method used: **IF IS (w, NOTHING) THEN**. In SIMPOL the same test would look like this: **if w =@= .nul**. Each of the object types listed above is described in detail in the "SIMPOL Language Reference".

A Word About Arrays

Arrays are typically found in most programming languages, but the version that is present in SIMPOL can be considered to be a superset of the functionality provided in most implementations. The first significant difference is that array is a type of its own. Also, most languages require that arrays be pre-specified to be of a specific size and type. Many automatically start at index 0, others allow a base index to be specified. In SIMPOL the array type is very flexible, although this also comes at a price.

Firstly, arrays are not required to be made up of only one type. It is perfectly acceptable to assign different data types to different elements of the array. This may not be a clever thing to do in all cases, but it is possible. Another interesting feature of the array type is that at each level of the array it is possible to have elements present. That means that it is possible to create a multidimensional array that looks like this:

```
array a,b
datetime dt

dt =@ datetime.new()
dt.setnow()

a =@ array.new()
b =@ array.new()

a[] = "Week Info"
a[1] = 10
a[1,0] = "Monday"
a[1,1] =@ dt
a[2] = 20
a[2,0] = "Tuesday"
a[2,1] =@ dt

b["array a"] =@ a
a =@ .nul
```

In the preceding example the majority of the capabilities of the array can be observed. Arrays can be indexed numerically or via strings or both. They can have values at every level of a multidimensional array, not just at the lowest level. Even the null element `a[]` can have a value or an object assigned to it. Any element can contain a value or a reference to an object. In the example above, the array `b` in element `b["array a"]` still contains a reference to the array object that was originally referenced by the variable `a` even after that variable has been set to `.nul`. As long as a reference to the object exists, the object itself still exists, and all elements that are associated with it.

As can be seen from the preceding example, it is possible to build quite complex arrays. Arrays can also be used in some cases in place of user-defined types. It is, of course, possible to create an array of user-defined type. The approach taken is left to the programmer, but it is strongly recommended that a consistent use is made of the array, since if they are used with varying data types it is entirely possible that an incorrect assignment may occur causing a type mismatch runtime error.

Note

It is important to realize that to detect if an array element is empty (as in not assigned), it is only necessary to test the value of the element in question. It is not an error to assign from an unassigned element. The value of any unassigned element is `.nul`. To get rid of an array element, it is necessary to set the element to `.nul`.

User-Defined Types

User-defined types are a significant advantage for any serious programmer, and can even be useful to less experienced programmers. At their simplest, user-defined types may consist of little more than a combination of value types that can be used together as a single unit. A perfect example of this might be a structure containing locale information. This type of structure would need to be passed to any function that is going to format a number, a date, or a time for presentation to the user and also to convert such data from the string representation provided by the user to an appropriate value or object type in the program. Here is what such a structure might look like:

```
type tLocaleInfo
  string sDecimalSep embed
  string sThousandsSep embed
  string sListSep embed
```

As can be seen from this example, it would be quite a bit more convenient passing around a single piece of information that contains all of the things that are important with respect to the locale than to have to pass each of these pieces of information around separately and to address them and store them separately as well. The example shown is a very simplistic implementation and does not include information about formatting dates or times, since these are also considerably more complicated than mere numeric formatting.

Type definitions must be located outside of any function, but they do not need to precede the function. The type definitions could be located in an include file and just be added on to the end of the program. Below is a gradual introduction to using user-defined types. Follow it through step-by-step and it should be failry clear at the end. The examples that include functions can even be tried out.

```
type mytype
  string m1
end type
```

In the type mytype, the string parameter contains a *reference* to a string object, it does not contain an actual string object.

```
type mytype1
  string m1 embed
end type
```

The mytype1 type contains an actual string object, not a reference.

To use the two types above see the following code:

```
function main()
  mytype m
  string s

  m =@ mytype.new()
  s = "hello"
  m.m1 =@ s
  s = "foo"
end function m.m1
```

This will return foo, since `m.m1` contains a *reference* to `s`.

```
function main()
  mytype1 m

  m =@ mytype1.new()
  m.m1 = "hello"
end function m.m1
```

This will return hello, since `m.m1` is an embedded string.

```
function main()
  mytype m

  m =@ mytype.new()
  m.m1 = "hello"
end function m.m1
```

This will result in an error, since `hello` is not an object, it is a value and this type can only hold a reference to an object.

```
type mytype2
  mytype mm1 embed
  mytype mm2
end type

function main()
  mytype2 m
  string s

  m =@ mytype2.new()
  s = "hello"
  m.mm1 =@ mytype.new()
  m.mm1.m1 =@ s
end function m.mm1.m1
```

This will result in an error 52, since the type mytype has not been defined as embeddable. The error not embeddable will be generated.

```
type mytype embed
```

```
  string m1
end type

type mytype2
  mytype mm1 embed
  mytype mm2
end type

function main()
  mytype2 m
  string s

  m =@ mytype2.new()
  s = "hello"
  m.mm1.m1 =@ s
end function m.mm1.m1
```

The above should now work as expected.

```

type mytype embed
  string m1 embed
end type

type mytype2
  embed
  mytype mm1
  mytype mm2
end type

function main()
  mytype2 m

  m =@ mytype2.new()
  m.mm1.m1 = "hello"
  m.mm2.m1 = m.mm1.m1
  m.mm2.m1 = .lstr(m.mm2.m1, 2)
end function m.mm1.m2
```

This should also work and produce an output of `he`.

```

type mytype
  string m1
  mytype next
end type

function main()
  string s
  mytype m,mfirst

  s = "hello"
  m =@ mytype.new()
  mfirst =@ m
  m.next =@ mytype.new()
  m =@ m.next
  m.next =@ mytype.new()
  m =@ m.next
  m.next =@ mytype.new()
  m =@ m.next
  m.next =@ mytype.new()
  m =@ m.next
  m.m1 =@ s
  m =@ mfirst
  mfirst =@ .nul

end function m.next.next.next.next.m1
```

This is an example of a singly-linked list which should return hello. The ability to include references to the same type as that being defined makes it possible to create complex data structures in memory, such as lists and trees.

A final note about embedded objects. Some objects cannot be embedded, such as fsfileinputstream or fsfileoutputstream or any of the ppcstype1 objects, mainly because they cannot be initialized by calling their new function. However, *references* to any object type can be part of a type definition.

The previous types consisted only of values and references to values but did not include methods. A more powerful kind of user-defined type is one that includes methods. Any user-defined type can also have a user-defined `new()` method that allows the programmer to do initialization of the newly created object when it is created. To create the methods, the functions must be defined in the same module (compilation object) as that where the type is defined and they must follow the type definition in the code file. It is defined by using the type name followed by the dot operator followed by the function name. The first argument to the function *must* be the type itself and in the case of the `new()` method it must return the object of the type that was passed in, otherwise the assignment to the variable will fail. See the example that follows:

```

type tCustInfo export
  string sCustID embed
  string sFirstname embed
  string sLastname embed
  datetime dtCreated embed
  string sCreatedBy embed
  function copy
end type

function tCustInfo.new(tCustInfo me, string sCreatedBy)
  me.dtCreated.setnow()
  me.sCreatedBy = sCreatedBy
end function me

function tCustInfo.copy(tCustInfo me)
  tCustInfo copy

  copy =@ tCustInfo.new(me.sCreatedBy)
  copy.sCustID = me.sCustID
  copy.sFirstname = me.sFirstname
  copy.sLastname = me.sLastname
  copy.dtCreated = me.dtCreated
end function copy
```

The preceding example shows a user-defined type that implements a `new()` method and a `copy()` method. The `copy()` method is implemented so that it produces an exact copy rather than a copy with a potentially new creator ID and new creation datetime. Typically such types will be defined and implemented in a single code file and then compiled as a SIMPOL pre-compiled module file that can be added to a project either at compilation or at runtime. That is the purpose of the export keyword in the type definition, to ensure that the type is visible outside the module. The functions do not require the export keyword since they are made available within the type.

If the `new()` method is listed inside the type definition then it can be called again to reinitialize the type. It is not necessary to list it, however, unless it should be possible to call at some point other than during the initial call to create the object.

Another important issue is the proper use of the keywords `embed`, `reference`, and `resolve`. By default, properties added to a type definition are references to items of a specified type (or any type using `type(*)`, any value type by using `type(=)`, or any matching tagged type when using `type(<tagname>)`). To make a property embedded, the `embed` keyword can be added to the end of the statement. To switch the default from by reference to embedded, the `embed` keyword can be placed inside the type definition on a line of its own. To switch back, place the `reference` keyword on a line by itself. These switches only apply within a type definition. The change of the default resets to "by reference" after exiting a type definition. The `resolve` keyword is used for a very special situation. Normally properties that are not embedded are not examined when trying to resolve the name of a property or method, but if the `resolve` keyword is added to the end of the property definition, then at runtime that property will be included when searching for a property or method that is not listed at the first level of the type definition. Let's look at a small example of this:

```
type myform
  form1 f
  string sFormname embed
end type

type myapplication
  myform mf resolve
  embed
  string sUsername
  datetime dtStart
end type

function main()
  myapplication app

  app =@ myapplication.new()
  #
  app.addcontrol(…)
  #
end function
```

Normally it would not be possible to call a method of the form1 object without directly referencing the f, but the use of the `resolve` keyword allows this. However, if the form1 object has not yet been initialized this will result in a runtime error number 21, "Object not found". Using the `resolve` keyword can help in creating powerful and easy-to-use types, but it is important that the types are designed in such a way as to minimize the likelyhood that those portions marked with `resolve` will cause an error because they are uninitialized. That might mean that the type's `new()` method takes parameters that allow the correct initialization.

Chapter 5. Operators and Expressions

Operator Overview

Most of the operators used in SIMPOL should look familiar to anyone who may have programmed in BASIC, C, C++, Java, or any of a number of programming languages. Some of the operators are specific to SIMPOL and need to be looked at more closely, in part because the very existence of these operators is a guide to effectively using and also to understanding the language itself. Just as it is essentially impossible to learn a human language without learning something of the culture that both formed and is formed by the language, a programming language embodies a specific approach to solving problems that may suit some people but not necessarily everyone, or it may embody an approach to a certain class of problem that is not as well addressed by other tools. The approach that is used in the language will then dictate the types of semantic devices that are necessary to support the creation of effective programs using the language.

The operators (and operations) in a language can be divided into a number of categories: assignment, arithmetic, comparison, logical, and object operators. Each of these will be discussed in detail in the sections that follow.

Assignment Operators

The standard assignment operator in SIMPOL is the equals symbol (`=`). This operator is used to assign a value to an object that is a value type. In other words, to assign a value to an integer, a number, a string, or a boolean object. This is equivalent to the use of the equals symbol in C and C++ and the assignment operator in Pascal and Delphi (`:=`). Unlike in most BASIC-derived languages, including the existing Superbase Basic Language (SBL) and Microsoft's Visual Basic, the equals symbol is not allowed to be used for both assignment and comparison. It is strictly used for assignment. See the section on comparison operators for more information.

As was discussed earlier in the chapter on data types, SIMPOL has two primary data types, value types and object types. For the value types the equals operator is used, since it is merely assigning a value to a variable. In the case of the object types, there is lot more going on, and it is important to realize that instead of a value being assigned to a variable, a *reference* to an object is being assigned to that variable. In SBL and Visual Basic, this is typically done using the SET keyword, as in:

```
DIM f AS Form
DIM c AS FormControl
SET f = Forms.Add("MyForm")
SET c = f.Controls.Add("tb1", "TextBox")
```

Rather than using a keyword, in SIMPOL there are two operators that can be used to make the assignment, either `@=` or `=@`. The example above converted into SIMPOL might look like this:

```
wxform f
type(wxformcontrol) c
f =@ wxform.new(...)
c =@ f.addcontrol(wxformedittext, ...)
```

This example is loosely based upon the current wxWidgets components and their data types. It would not actually work in SIMPOL as it is written unless the method calls were filled out with all of the relevant parameters.

Arithmetic Operators

The usual set of arithmetic operators are also included in SIMPOL, such as: addition `+`, subtraction `-`, multiplication `*`, division `/`, modulus `mod`, and negation (unary minus) `-`. They are used in the usual way, but have a few interesting points when applied to the string data type. For details see the appendix. Just to provide a few examples, however, if a string is multiplied by an integer then the result is the integer's value copies of the string. Subtracting a string from another string results in a string that has had all of the substrings removed that match the argument that was being subtracted.

Comparison Operators

The list of comparison operators consists of symbols that should be familiar to most programmers, regardless of whether they are C or BASIC oriented. The operators currently supported are:

- Equal to (`==`)
- Greater than or equal to (`>=`)
- Less than or equal (`<=`)
- Not equal to (`!=` and `<>`)
- Greater than (`>`)
- Less than (`<`)

The `==` symbol, although familiar to Java, C, and C++ programmers is a bit of a new experience for BASIC programmers. Consider this symbol to be merely an unambiguous method of separating assignment from comparison. There are also two different symbols for *not equal to*, one used commonly in Java, C, and C ++ and one found commonly in BASIC-oriented languages.

The set of operators in the previous paragraph are meant to be used with value types. For comparing object types, there is a different, more limited set of operators. This is mainly because object types are more complicated and must be compared in different ways and also because object types can have a value associated with them. As an example, date types where the *value* of the object is an integer equal to the total number of days in the date since Jauary 1, 0001. This feature of the language required a different set of operators to be established for the comparison of object types, again to be unambiguous. These operators are listed below:

- Refers to the same object (`=@=`)
- Does *not* refer to the same object (`!@=` and `<@>`)

In some languages, such as in SBL, there is a function to perform the comparison of two object variables. In SBL this is done with the `IS()` function. In keeping with the decision to try and limit the number of keywords in the language, it was decided to use operators for this purpose rather than add keywords. It is important to understand the difference between these operators and the ones used for comparing values. Look at the following example:

```
function main()
  string s1, s2, sResult

  s1 = "foo"
  s2 = "foo"
  if s1 == s2
    sResult = "They are equal in value"
  else
    sResult = "They are not equal in value"
  end if

  if s1 =@= s2
    sResult = sResult + " and they refer to the same object."
  else
    sResult = sResult + " and they refer to different objects."
  end if
end function sResult
```

When this program is run it will output as its result, `They are equal in value and they refer to different objects.` It is also possible to have a string variable refer to the same object as another string variable. In that case, any change to the value of the first variable will also change the value for the second since they both refer to the same object. See the example shown below:

```

function main()
  string s1, s2, sResult

  s1 = "foo"
  s2 =@ s1

  s1 = "foobar"

  if s1 == s2
    sResult = "They are equal in value"
  else
    sResult = "They are not equal in value"
  end if

  if s1 =@= s2
    sResult = sResult + " and they refer to the same object."
  else
    sResult = sResult + " and they refer to different objects."
  end if
end function sResult
```

When this program is run it will result in the output, `They are equal in value and they refer to the same object.`

Logical Operators

The set of logical operators comprises the `and`, `or`, and `not` operators and in this case, they are all keywords rather than symbols. It is important to understand that these operators are *only* logical operators, they are not bit-field operators! The return value of any logical operation will be one of either `.true`, `.false`, `.nul`, or `.inf`.

Bitwise Operators

The set of bitwise operators comprises the `AND`, `OR`, and `XOR` operators and in this case, they are all keywords rather than symbols. It is important to understand that these operators are *bit-field* operators, they are not logical operators! For details of the operators and their values see the "Bitwise Operators" section in the "SIMPOL Language Reference".

Warning

The bitwise operators and the logical operators should not be mistaken for each other! They can have very different results from what is expected if used in the wrong way. Remember, SIMPOL is a *case-sensitive* language.

Object Operators

Object operators are represented by the property operator also known as the dot operator (`.`) and the member operator also known as the shriek, bang, or exclamation point operator (`!`). The property operator is used to access the properties and methods of an object. This is similar to the way it is used in numerous other languages. The member operator is used to access member information in a related member of the object in a way that is specific to the data type in use, athough it is similar to accessing a member of a collection in other languages. The best way to illustrate the use of the member operator is with a few examples:

```
function main()
  integer iErrnum
  ppcstype1 ppcs
  ppcstype1file f
  ppcstype1field sfldLastname
  ppcstype1record r
  boolean bFound
  string sResult

  // Initialize iErrnum so that it refers to an object rather than
  // to .nul
  iErrnum = 0
  // Initialize ppcs to use a port and act as a user called test
  ppcs =@ ppcstype1.new(udpport=1289, error=iErrnum, \
                        username="test")
  // If the initialization succeeded ...
  if ppcs !@= .nul
    // Open the file CUSTS at www.superbase.co.uk on port 1280
    f =@ ppcs.openudpfile("www.superbase.co.uk:1280", "CUSTS", \
                          error=iErrnum)

    // If the file opened successfully ...
    if f !@= .nul
      // Retrieve a reference to the field in the CUSTS file whose
      // name is Lastname. Please note that if there is no field
      // called Lastname this will result in an untrappable error.
      // If there is any concern that a field may not be present,
      // it would be better to use the function getfield() from
      // the db1util.sml library file since that will return .nul
      // rather than causing an error. This reference in both
      // cases is case-sensitive. If the field contains any
      // characters that are not valid in an identifier then it
      // should be placed in double-quotes. Variable references
      // are not permitted as the argument following the member
      // operator.
      sfldLastname =@ f!Lastname

      // Initialize the bFound variable to refer to an object
      // instead of .nul
      bFound = .false

      // Assign the results of the lookup of the value Smith in
      // the Lastname index to the r variable. Since we passed the
      // error and found objects in the function will return the
      // nearest record even in the case of an inexact match.
      r =@ sfldLastname.index.selectkey("Smith", error=iErrnum, \
                                        found=bFound)

      // Test that the r variable points to an object (if the file
      // were empty it would return a .nul object
      if r !@= .nul
        // Assign the value contained in the ppcstype1record
        // object referred to by r that is referenced by the
        // Lastname field of the file object. This can also be
       // done by using the get() method of the record object.
        // Again, if there is no field called Lastname (case-
        // sensitive) in the file then this assignment will result
       // in an untrappable runtime error.
        sResult = r!"Lastname"
      end if
    end if
  end if
end function sResult
```

As can be seen from the example above, there are different operations taking place when the member operator is used depending upon the type with which it is used. In the first case, the argument to the member operator is used to lookup a field name in the ring of fields and to return a reference to a field object which must be assigned using the `@=` operator. In the second case, a much more complex operation is taking place. The argument to the member operator is being used to lookup a field reference in the file object reference that is part of the record object and that is then used as an argument to the `get()` method of the record object.

In every case, the member object is used to provide a type of shorthand that results in a logical assignment of what otherwise might be a number of programmatic steps. As was stated in the remarks in the example, any error will result in an untrappable runtime error that will halt the program. Also, the overhead for using this approach is normally higher than using the more mundane approach and in some cases may need optimization using the alternative method if the section of code is too slow. That is because this requires a lookup each time rather than doing the lookup once and storing the result, so in a loop the cost of doing the lookup over and over again can make itself felt.

Expressions and Statements

Expressions are the building blocks of a program. Variables and operators are combined together to produce a result. An expression can be extremely simple or exceedingly complex. Simple expressions consist of two variables, a variable and a constant, or two constants that are added, subtracted, etc. that produce a value. Below are some examples of expressions:

```
x + y
3 * x
s + "hello"
(3 - z)/((x + y) * 9)
```

A statement is made up of one or more expressions and accomplishes something. It is considered to be a complete unit of grammar within a programming language and must be ended with an end-of-statement character. In SIMPOL the end-of-statement character can be the end of the line, the semi-colon (;), or the colon (:). As can be seen from the expressions in the example above, there is no result that occurs, regardless of how complex the expression is, since in no case is the expression being assigned to something or the result of the expression being used in some way.

Statements come in a number of varieties: assignment statements, if statements, while statements, and function calls. The number and variety of statements in a programming language is directly related to the number of keywords to be found in that language. In SBL there is very large number of keywords and a thus a proportionally large number of different statements: `MENU`-, `ADD FORM`-, `ADD DIALOG`-, and `SET`-statements and numerous others, often with cryptic parameters in varying combinations. The level of complexity in learning a language is directly proportional to this. In SIMPOL there is a very small set of keywords and therefore a similarly small set of statements. Complexity is added by adding objects, but even there, a great deal of attention has been paid to ensuring that the objects are extremely consistent in their design and have methods and properties in common wherever it would make sense to do so. Below are some examples of statements:

```
z = x + y
f.amount = 3 * x
if s + "hello" == "othello"
while (3 - z)/((x + y) * 9) > 0
foo(z2)
```

In each of the statements in the example above, the expression is being either assigned, evaluated, or is a call to another function.

Chapter 6. Statements and Flow Control

At the end of the previous section, we discussed expressions and statements. In this section we will go into how statements are used to build functions and how functions make up a program.

`function`

The function is the basis for every program in SIMPOL. The simplest program consists of a single function called `main`. When the `main` function is exited, the program also ends. A function begins with a function statement. The function statement consists of the function keyword, followed by the name of the function which *must* be a valid identifier, followed by the left parethesis, followed by zero or more parameters in the format `type identifier` white space `parameter name` optionally followed by an equals sign and a default value for the parameter. Multiple parameters are separated by commas. The parameters are then followed by a closing right parenthesis. If the function is part of a library and should be exported, then the `export` keyword follows the closing parenthesis. The complete syntax diagram can be seen below:

`function` functionname `([typename` parameter `[=value]] [, typename` parameter `[=value]] [, …]) [export]`

Tip

One point worth noting is that in the function declaration there is no indication of whether or not there is a return value, nor if there is one any information about its type. That is mainly because the return value follows the **end function** statement and the type may not be known when the function is written or even when it is compiled.

`if`

The `if` statement in SIMPOL is similar to that in most languages. There are some differences with the `IF` statement from SBL, specifically that there is no `THEN` component and also no concept of a one-line `IF` statement that does not require an `END IF` statement. In SIMPOL every `if` statement requires a matching `end if` statement. Otherwise the `if` statement is equivalent to that in SBL and in other BASIC-derived languages. There is also an `else if` statement and an `else` statement as optional parts of the `if` statement. The syntax diagram follows:

`if` <expression> ;|:|newline <statement> ;|:|newline [`else if` <expression> ;|:|newline <statement> ;|:|newline] [`else` <expression> ;|:|newline <statement> ;|:|newline] `end if`

Only one `else` statement can exist and it must be last, but multiple `else if` statements are allowed.

`while`

The `while` statement is a very useful construction and is the primary tool for creating loops. It is very flexible since it can have a start condition, an end condition, or both start and end conditions. There is no method of breaking out of a `while` loop. In keeping with the basic design of SIMPOL, there is one entrance and one exit to the `while` loop.

Tip

SIMPOL doesn't have a `for ... next` loop nor does it have a `repeat ... until` loop. Instead the `while ... end while` loop is used for these cases. The `for` loop is a subset of a `while` loop in that it automatically increments the loop variable a specified amount. Since this is just a special case of a `while` loop, it was not added. In a compiled language there is no advantage, even if a `for` loop were to have been provided, it would have compiled to the same code as a `while` loop. As for the `repeat ... until` loop (or the `do ... while` loop) that is equivalent to using the SIMPOL `while` with no starting condition and with an ending condition.

The basic `while` loop looks like this:

```
integer err, i

// Basic while loop (similar also to for...next loop), no ending
// condition
i = 10
while i > 0
  i = i - 1
end while

// Repeat...until style loop, no starting condition
i = 10
while
  i = i - 1
end while i == 0

// Both conditions in use, the starting condition tests the loop
// variable and the ending condition tests the error return value
i = 10
err = 0
while i > 0
  err = testfunc(i)
  i = i - 1
end while err == 0
```

The preceding example shows three different uses of the `while` loop. An important point to consider is that the ending condition following the `end while` keywords should be read as "end the while loop if the condition is true".

Tip

Please note that the condition must evaluate to either `.true` or `.false`.

Chapter 7. Variables

Variables are placeholders that are used in a program in order to provide a method of accessing the objects that are acted upon by the program. They are really like the glue that holds everything together. In this chapter we will discuss how variables are used in a SIMPOL program. We will discuss the various types, how to create them, their visibility within the program, how some variables can hold more than one type of object, and how variables affect the objects they represent.

Variable Typing

SIMPOL is what is known as a *strongly typed* language. By this is generally meant that it is considered an error to assign an object or value of one type to a variable of another type. As an example, if I have one variable that is declared to be of type `integer` it is an error to assign a variable of type `number` to that variable. It will result in a type mismatch error. This is one way that the programming language protects the programmer from making an error that might otherwise be very hard to find. If instead of generating an error, the programming language automatically converted the variable of type `number` to an integer value, truncating or rounding the non-integer portion of the value, it would be very difficult to track down, especially in a large program since the problem may only appear to be intermittent (it would only occur when the value in the variable of type `number` was not an integer value).

That is all well and good, but there are some situations where it is absolutely essential to be able to handle more than one type using only one variable. As an example, consider the situation where you may wish to process all of the controls on a form. Each form control has its own type. If it is possible to declare a variable to be of type `FormControl` and if that type is designed to represent *any* form control, then it would then be possible to use a single variable to contain a reference to any control on the form, without causing an error. In SIMPOL by using the `type` property of the object it would then be possible to detect which type the variable currently contains and to perform appropriate operations on that object. Later in this chapter this capability, also known as polymorphism, is discussed in greater detail.

Declaring Variables

Variables in SIMPOL *must* be declared before they can be used. They also do not carry any sort of type designator, as is common in various dialects of BASIC, such as a dollar sign for strings or a percent symbol for integers. Currently there is only one method available for declaring variables. A program statement must begin with the type designator and be followed directly thereafter by the variable name.

```
function main()
  string s
  s = 'foobar'
end function s
```

In the preceding example the variable `s` is declared to be of type `string` before the text value `foobar` is assigned to it. One of the more interesting things that this syntax allows is to declare a variable to be of a type that may be unknown to the programmer at the time that the program is written. This can occur when a database field is passed to a function and a variable must be declared to be of the same type as the contents of the field. The example that follows demonstrates this:

```
function display(ppcstype1record r, ppcstype1field fld)
  string s
  fld.datatype temp
  if temp.type == string
    s = r.get(fld)
  else if temp.type == integer
    s = .tostr(r.get(fld), 10)
  end if
end function s
```

In this example the variable `temp` is declared to be of the same type as the datatype of the field. Each time the function is called it may be passed a field with a different content type. Similarly, a function could return a different value each time it is called, if the return value is dependent on one of the parameters passed and the return value is declared by using the datatype of one of the parameters that is passed into the function.

Although these capabilities can provide considerable flexibility and power when designing programs, it is also possible in even a medium sized program to lose track of the type of a variable, especially if that variable is dependent on the datatype of a database field. It is therefore strongly advised that in any larger function that some sort of naming convention be adopted for naming variables. It isn't necessary to make them as complicated as the notation often associated with Windows C programmers. Since there is no limit to the size and precision of integers and numbers in SIMPOL and no significant pointer capability, it usually sufficient to indicate the type with a single letter. one convention that may show up regularly in the supplied examples is to use a lowercase letter to indicate the type, then an uppercase letter and then the remainder is lowercase or in some cases title case where words are joined together. Generally we use: s for string, i for integer, n for number, b for boolean, d for date, t for time, dt for datetime, fsi for fsfileinputstream, fso for fsfileoutputstream, r for record, ppcs for ppcstype1, etc.

There is no concept of the `REDIM` keyword in SIMPOL. If a variable is declared at one place in the function, and then there is a new declaration using the same variable name at another place in the function (even if the type changes), this is not an error. The variable is considered to be destroyed at that point and a new variable is created of whatever type designation has been used in its declaration. This feature can, however, lead to errors that may be hard to detect, so it is important that the programmer be cautious in their use and reuse of variables. There is no advantage to the compiled program of using one variable name twice or two different variable names. From the point of view of program maintenance, it may be better practice not to use this feature unless it is abundantly clear from the program why it was used.

Another important point to remember is that variables in SIMPOL do not automatically initialize to zero or the empty string. The initial value of any variable is `.nul`. This may cause some confusion at the beginning since any operation that includes a value that equals `.nul` is also going to equal `.nul`. *Always remember to initialize any variable before using it!* Also, if a variable is not initialized before it is passed to a function, then the local variable in the function will also be equal to `.nul` until a value is assigned to it. More importantly, since there is no object to which to assign the results when the function returns, nothing can be passed back to the calling function in that parameter.

Variable and Type Scope and Visibility

Scope and visibility are often a complex topic in programming languages. That is not the case with SIMPOL. In SIMPOL there is only one kind of scope and two kinds of visibility. Before we get into the details, however, it may be useful to explain what these two concepts actually mean. By scope, we generally mean the area of the program where a variable is still in existence and is accessible. BASIC derived languages often have two or more types of scope, global and local being the most common.

Global scope means that the variable is visible and accessible anywhere in the program. It also means that the variable will not be destroyed until the program ends or some statement within the program expressly destroys the variable. Globally visible and accessible variables are often the root of unidentifiable side-effects in complex programs. In a programming language like SIMPOL that is multi-threaded, allowing global variables would be extremely messy, since they would have to be visible in every thread and may change unpredictably depending on how the various threads are scheduled and executing. The alternative would have been to add syntax to lock them which would have added overhead and complexity. There are no global variables in SIMPOL.

Local scope often means within a function, although in some languages it may be only within a block statement, such as a for…next loop that is itself within a function. Local scope in SIMPOL means within a function. From the point in a function where a variable is declared it is visible and remains in existence until the function ends. When the function ends, all of the variables are destroyed, any memory they are using is released and it is as if they had never existed. Variables are not visible outside of a function. although they can be passed as arguments to another function. Technically though, once the function is entered a new local variable is created and the value of the variable in the calling function is assigned to the new local variable which is then in scope until the end of the function at which point its value (which may have changed) is then reassigned to the variable from the original calling function. Static variables are a special form of local variable that retains its value when the function is exited but is only accessible from within the function. There are no static variables in SIMPOL.

Visibility is similar to scope but generally is used to refer to the ability to access type definitions and functions. As described earlier, there are two kinds of visibility in SIMPOL, global and modular. All of the intrinsic types and functions are globally visible. User-defined types and functions are visible only within the same compiled unit unless they have been expressly made globally visible by exporting them using the `export` keyword. Typically if a program is made up of a main code module plus some linked in libraries (whether self-made or from other source) then the code libraries will make some of their types and functions visible for use by other programs but they may not make all of the types and functions visible unless that is necessary to use the library. There may be only one interface function that is exposed but in actuality there may be a dozen or more functions in the module that are used to implement that exposed function. By only exporting the interface function, the programmer can reduce the level of error checking on the implementation since they don't need to worry about those functions being called from outside the module.

Value Types, Reference Types, and Type Tags

There are two conceptually different data types within SIMPOL, value types and reference types. Value types are, as the name implies, associated with values. These types are similar to the scalar types in other languages. Variables that are declared as: `boolean`, `integer`, `number`, or `string` are value types. When a variable is declared to be of one of these types and a value is then assigned to the variable, an object is created and associated with the variable and the value of the object is set to the value that is being assigned. Values can be assigned to variables of this type using the `=` operator.

Reference types are more complicated, since they do not merely contain a single value but represent more sophisticated objects. An object is assigned to a variable of this type using the `=@` or the `@=` operator. This is similar to the construct common in various BASIC dialects including Microsoft Visual Basic and Superbase Basic Language that uses the `SET` keyword and the equals symbol.

There are a couple of important points to realize when working with reference types. First, even value type variables can be used as reference variables. Second, more than one variable can refer to the same object. See the example below:

```
function main()
  integer i
  integer j

  i = 1
  j =@ i
  i = 3
end function j
```

The statement **i = 1** assigns to the integer object referred to by `i` the value of the integer constant `1`, whereas **j =@ i** causes both `i` and `j` to refer to the same integer object, so that when a value is assigned to `i` it is setting the value of the object to which `j` refers. This applies to any reference type and can provide a great degree of flexibility when writing programs. As an example, a database record is represented by a single variable and a second variable can easily point to the same database record while potential modifications are happening to the first variable. If those changes are occurring to the actual object, then the second variable will also be aware of the changes. If the first variable is then reassigned to another object, the second variable will still refer to the original object.

So what are *type tags* and why would anyone want to use them? Earlier in the chapter we discussed the usefulness of having a variable be able to refer to objects of more than one type. In a strongly typed language like SIMPOL, this normally wouldn't be possible. The `type` object has two functions that are accessed by the convention **type(*)** and **type(=)**. The first of the two is used to declare a variable that can contain a reference to any type. The second is used to declare a variable that can contain a reference to any value type. That sounds pretty useful, we can now declare a variable that can refer to any type, so why use anything else? Mainly because it is considerably more expensive to handle a variable that can hold a reference to any type and also because it makes it very difficult to find errors in the program.

In line with that kind of thinking, type tags (you knew we would get back to them sooner or later) were introduced to allow the declaration of variables that could refer to only a limited set of types. So how does this work? Imagine we are creating a group of types to represent form controls. We may create a text box object, a check box, a command button, and so on. We might choose to assign a tag to each of the types called `FormControl`. By doing that we can then use the type object to create a variable that can refer to any type that is tagged as `FormControl` but not any other type, so if there is a mistake in the program it will still break at the right point for the right reason. The way we declare the variable looks like this: **type(FormControl) fc**. So how do we actually assign the tag? Look at the following example:

```

type tTextBox (FormControl, EditControl)
  string Text embed
  boolean Enabled embed
  type(FormControl) next
end type

type tCheckBox (FormControl)
  string Caption embed
  boolean Enabled embed
  boolean Selected embed
  type(FormControl) next
end type
```

In the preceding example the `tTextBox` type is tagged as being both a FormControl and an EditControl. The `tCheckBox` type is only tagged as a FormControl. A variable that has been declared to be of type tag FormControl can hold a reference to either of these two types. Before we leave type tags behind us, it is important to point out that a local variable can be declared either inside the function or else in the parameter list of the function.

Variable and Object Persistence

"Variables are like the glue that holds everything together." This description is especially appropriate in SIMPOL, since any object that is no longer referred to by any variable anywhere within the program will immediately be discarded. This is an important point to understand. If an object is no longer referenced in any way by the program, via a variable or a property of an object that is itself referenced by a variable it will be discarded. Even if there is a linked list of objects each of which refers to the next, as long as the beginning of the list is anchored by being referred to by a variable the entire list will still exist. Once there is no way for the program to refer to the beginning of the list, any object not referred to by a variable will be discarded. If the third element of the list is still referred to by a variable but the base is not, then the base and all elements preceding the third member will be discarded. If, however, each object in the list has a property that refers to the preceding object as well as one that refers to the next object (a doubly-linked list) then as long as *any* member of the list is referred to by a variable (or by another object that is anchored by a variable) then the entire list is safe and will not be discarded.

This allows for the creation and use of quite complex data structures in memory while only retaining a single base variable to anchor the entire structure. Once a function is exited, all local variables created within the function are destroyed. If the local variable were in the parameter list, then the corresponding variable in the calling function will be assigned the value of the local variable prior to the variable being destroyed.

An important point to remember is that if a variable is passed to a function and the variable has not yet been initialized to refer to an object, then it cannot receive any changes made within the original function since no object exists to assign the results to. Also, it is not possible to create an object in a function and assign it to a local variable and then have that object returned to the calling function. The only way to do this is to have the new object be the return value of the called function and to assign the results of the function call using the object reference assignment operator (`=@`).

Chapter 8. Intrinsic Functions

The Nature of Intrinsic Functions

In SIMPOL intrinsic functions are defined as functions that are always available (that are part of SIMPOL). They always begin with a dot (.), take a constant number of parameters (although the data type of the parameters may not be fixed), have no named parameters (so all parameters must *always* be specified), operate only on values and they always return a value. Also there is no function object to represent an intrinisic function. The dot operator preceds the function name to ensure that no user function can be defined that would conflict with a current or future intrinisic function (user functions cannot be defined with a name that begins with the dot operator).

The remainder of this chapter is divided into sections that briefly describe the various intrinsic functions grouped under a specific heading. The various types of intrinsic functions can be roughly grouped under the following headings:

- Compression Functions
- Conversion Functions
- Numeric Functions
- Selection Functions
- Blob Functions
- String Functions

The name of each group describes the type of functions that it includes. As time passes, the list of intrinsic functions will undoubtedly grow and quite possibly additional groups will be added as well. When that happens this section will be updated.

Compression Functions

SIMPOL provides some basic compression and decompression functions for compressing strings and blobs. Currently there is only one of each type, listed below:

- `.compress1()`
- `.decompress1()`

The functions will normally be supplied in pairs. The names of the current set end in the digit 1, primarily to make clear that they are not the only version nor are they very likely to be the last version as well as to associate them with each other. For the actual usage details see the Intrinsic Compression Functions section in the "SIMPOL Language Reference".

Conversion Functions

The conversion functions group includes functions that are used to convert from one value type to another. Whether converting from string to integer or number, from one of the numeric types to string, or even converting from a blob to a string, the functions will be classified as conversion functions. The list of the current intrinsic functions from the conversion group is:

- `.char()`
- `.charval()`
- `.deintegerize()`
- `.integerize()`
- `.lcase()`
- `.tcase()`
- `.toblob()`
- `.tostr()`
- `.toval()`
- `.ucase()`

The details of the proper syntax and usage of each of these can be found in the "SIMPOL Language Reference" in the "Conversion Functions" section of the "Intrinsic Functions" chapter.

Numeric Functions

Numeric intrinsic functions are specific to working with numeric values, whether they are integers or numbers. They are generally used to perform some mathematical operation using the value or values passed. The following is the current list of numeric intrinsic functions:

- `.fix()`
- `.ipower()`
- `.ipowermod()`

The details of the proper syntax and usage of each of these can be found in the "SIMPOL Language Reference" in the "Numeric Functions" section of the "Intrinsic Functions" chapter. Of the three, the `.fix()` function is the most useful for most people. The other two are primarily used in the implementation of RSA encryption.

Selection Functions

Intrinsic selection functions as a group includes those functions that are used to make a choice from among the arguments and then to return one of them. The list of functions in this group is:

- `.if()`
- `.min()`
- `.max()`

The simplest of these is the `.if()` function, which evaluates the first argument and then if it is equal to `.true` it returns the second argument otherwise it returns the third argument. The other two functions: `.min()` and `.max()` return either the item with the lowest value or the highest value respectively. For full information regarding the proper syntax and usage of each of these see the "Selection Functions" section of the "Intrinsic Functions" chapter in the "SIMPOL Language Reference".

Blob Functions

The blob data type requires certain special functions to cater for the ways in which it will be manipulated. The functions currently available are:

- `.inblob()`
- `.subblob()`

The first is used to find the first matching blob in another blob. The second is used to extract a blob from a blob beginning at some offset for a specified length. These two functions are comparable to the string functions `.instr()` and `.substr`. For further information about these two functions see the "Blob Functions" section of the "Intrinsic Functions" chapter in the "SIMPOL Language Reference".

String Functions

There is a special set of intrinsic functions for working with strings, just as there are for blobs. The following list contains all of the string-specific intrinsic functions:

- `.instr()`
- `.len()`
- `.lstr()`
- `.rstr()`
- `.substr()`

The first of the functions is used for finding a match for a string within another string. The second returns the length of the string in characters (not bytes!), and the last three are for slicing up a string; `.lstr()` returns a string beginning at the first character for the desired number of characters, `.rstr()` does the same starting from the end of the string and working toward the beginning, and `.substr()` takes a starting point and a count and works from left-to-right to return any substring from any point in the original string. For the precise technical description of these functions, see the "String Functions" section of the "Intrinsic Functions" chapter in the "SIMPOL Language Reference".

Chapter 9. System Functions

The Nature of System Functions

System functions differ from intrinsic functions in several ways. They are allowed to have named parameters (or not), the parameters can have default values, and not all parameters are required. To differentiate between the intrinsic functions and the system functions, the former begin with a dot and the latter with an exclamation mark. In both cases this has been done to ensure future compatibility of code. New functions that are added to the language as part of core SIMPOL will have either the dot or the exclamation mark at the beginning and will therefore never be able to have the same name as a user-defined function. This chapter discusses the various system function that are part of SIMPOL.

The `!beginthread()` Function

SIMPOL provides a multi-threaded program execution environment regardless of whether or not the target development platform implements support for multiple threads. Writing programs to use multiple threads can generally be considered an advanced topic, but in SIMPOL it is fairly straightforward. All that is necessary to start a new thread is to call the `!beginthread()` function passing the name of the function where execution should begin. A second optional parameter allows the user to pass a reference to any type. This will most often be some type that provides some additional information to the function that would otherwise not be available. When a new thread is begun the original thread continues execution without waiting for the results of the new thread. In SIMPOL a common use of multiple threads can be found in the way that the tcpsocketserver type works. Each time a connection is made to the server, a new thread is created that begins execution at the function that is passed in the `listen()` method. Each of the threads executes concurrently with the others and with the original server thread.

Another place that will see regular use of threads is the user interface support provided by the window1 type. For each window that is visible (including child windows but not including form controls) a separate thread is required to manage the events for that window. If no thread is provided then the window will appear to be dead, because it will not respond to events.

The first parameter to the function is a reference to the function that should be called. In practice, this will simply be the name of the function but it could also be a function variable that has the function assigned dynamically during program execution. The second parameter is optional, a reference to any type. This parameter is very important since it is the only way to provide access to information about the remainder of the program that may be needed within the function. Remember there are no global variables in SIMPOL.

The biggest problem with having multiple threads all having access to the same object at the same time is when more than one thread wishes to change the value of some property within the object. This is not generally safe since there is no way of knowing which thread is doing what at what time compared with the others. The only safe way to modify values would be to lock the object or some portion of the object. This can be done using the lock1 type. It is provided specifically to facilitate the safe use of common objects by multiple threads concurrently. If one thread wishes to modify a value in a common object it can attempt to lock the lock1 object that is a member of the common object. If that fails, then it needs to wait and try again (retries can also be built into the call to lock). This allows for the safe regulation of access to the common object. For more information about the lock1 type see the "SIMPOL Language Reference".

The `!loadmodule()` Function

The capability to create libraries implies an ability to not only link the libraries at compile time but also to load a library dynamically at runtime. This function provides that capability. It has limited use in most cases currently because there is no capability for detecting loaded modules nor is there a method for unloading them. These capabilities are planned for the future.

The `!wait()` Function

This function provides are system friendly method of waiting for a specified amount of time. The argument to the function is the number of microseconds that the function should wait. While the function is waiting, it will not make unnecessary use of system resources, however, if there is more than one thread being processed, a true wait will only occur if all of the threads are waiting for some reason (waiting for a connection, waiting via this function, waiting in some operation that includes a retry and timeout, etc.).

Chapter 10. User-Defined Functions

User-defined functions are one of the most important characteristics of any modern programming language. They provide the programmer with the ability to write modular programs and create reusable code components. Over the course of time a good programmer will build up a powerful toolbox of regularly-used functions that have been well-tested. This toolbox of functions enables the programmer to produce powerful and reliable programs quickly and easily.

Defining and Calling Functions

To define a function the keyword `function` is placed at the beginning of the line and outside the body of any other function. That is followed by at least one space and then the name of the function. Following the name is an opening parenthesis, the parameter list and then the closing parenthesis. The parameter list may be empty. If it is not, then the parameters are listed starting with their type and then the name of the parameter. A default value can also be assigned to a parameter by placing an equals sign after the parameter. The entire syntax diagram would look like: `function <name>([<parameter type> <parameter name>[=<value>],…])`. The return value of the function is placed on the last line of the function following the `end function` statement. This can be a variable, an expression, or even the entire function body (assuming it is a single statement). A function does not need to return a value.

To call a function it is sufficient to place the name of the function followed by the open parenthesis, the argument list, and the close parenthesis in the program code. If the function returns a value that can be assigned to a variable or used within an equation. Even if a function returns a value that value can be ignored if the programmer so chooses. The following is an example of the definition and calling of a function:

```
function main()
end function hello("world")

function hello(string s)
  string t

  t = "hello " + s
end function t
```

Function Scope

A user-defined function is only visible within the unit in which it is compiled unless the function was also defined with the `export` keyword. This makes it possible to create reusable code modules compiled as SIMPOL library files (*.sml) and to only expose the functions that represent the interface. All of the supporting functions that are not exposed are invisible to external callers.

Function References (Pointers)

SIMPOL supports the concept of function references, so it is completely permissable to declare a variable of type function, then assign a reference to a function to the variable, and finally use the variable to call the function that is referenced. This provides a powerful mechanism for writing generic code that can allow function references as parameters enabling functions to be written that pass off the responsibility for certain operations to functions that are defined by the caller. The caller can pass these functions as references. An traditional example of this would be a sort function that takes a comparison function as a parameter. The sort function only needs to be able to manipulate the contents of the array, it does not need to know how to compare the members.

Part III. Web Server Applications — CGI, ISAPI, and FastCGI for Dynamic Web Content

In this part we will cover using SIMPOL to generate dynamic web pages and truly powerful web-based applications using a variety of techniques all based on creating web content at the server. Using the technology described in this part, it is possible to build some fast, powerful, and reliable server-side web applications.

Table of Contents

Chapter 11. SIMPOL Web Server Applications

Introduction

One of the powerful features in the new SIMPOL language is the built-in support for producing web server applications. The key to this is the cgicall type and the various loader programs for supporting this type. SIMPOL supports several standard ways of working with this type, which provides access to the Common Gateway Interface (CGI). This includes standard CGI, ISAPI (Internet Information Server API), and Fast-CGI. All of these technologies work in similar ways and ISAPI and Fast-CGI are based on the older CGI technology.

A significant difference between the ISAPI approach and traditional CGI is that the ISAPI server extension is normally loaded once and then left loaded, whereas the standard CGI program is loaded and then unloaded each time it is called. The advantage, especially in the case of an interpreted or byte-code compiled program is that the interpreting environment is left loaded and only the program must be loaded and run each time. This is similar to Microsoft's ASP (Active Server Page) and Sun's JSP (Java Server Page) technology in that the interpreters for these technologies are built in or else dynamically loaded by the web server enabling them to provide improved performance. One difference in this is that SIMPOL programs are not combinations of code and HTML markup, but are instead compiled programs. The difference in performance can be considerable. The ease of design of ASP and JSP pages is also supported within the SIMPOL IDE. Using the server page support SIMPOL source code can be mixed with HTML on the same page in exactly the same style as in ASP pages. When the project is compiled, the server page is also compiled, providing the best of both approaches — mixed-mode design plus the speed of compiled code.

If performance is really what you are looking for though, then the real answer is Fast-CGI. Using the Fast-CGI support in SIMPOL it is possible to not only load the execution environment and leave it loaded, it is also possible to load the actual program and allow it to perform its initialization once and then thereafter only respond to calls. This approach is the fastest that is realistically possible (short of adding a special program in a compiled language like C directly to the web server code). A SIMPOL Fast-CGI program has an initialization function, an execution function, and a termination function. The initialization function is only called the first time the program is loaded. The execution function is called each time a call is made to the program by the web server and the termination function is called only when the program is being unloaded. Fast-CGI is currently supported by a number of web servers on various platforms, most notably though by the Apache web server.

All of these technologies are very unified in the way that they are implemented in SIMPOL. In each case a parameter of type cgicall is passed to the starting point in each program. In each case the program can then act in a similar fashion, retrieving server variables using the `getvariable()` method or key values from a posted form using the `keyvalue()` method. In fact, with careful design it is possible to write a program that will work in all environments without change. Such examples can be found in the examples included with the product. One item of good design that is used consistently throughout the examples is that of storing parameters that might change in a configuration file and retrieving them at runtime. This means that the program does not need to be recompiled or modified to run in different locations or even on different platforms.

How it Works

To produce a program that can be called from a web server is fairly easy. Below is a standard web version of a "Hello World" program that is written in SIMPOL:

```
function main(cgicall cgi)
  string s
  s = "Content-type:  text/HTML{d}{a}{d}{a}"
  s = s + "<html><body>Hello World!</body></html>"
end function s
```

This program might be saved as `cgihello.sma`. When compiled the program would normally be called `cgihello.smp`. To get the Apache web server to run this program it would have to be prefaced by what is known as a *shebang* line. This is a convention that originated on Unix. It is formatted in such a way that it is normally considered a comment in shell scripts but this one has a special format and is interpreted to determine which program should be used to execute the remainder of the script. In the case of a web server program this line is retrieved by the web server and used to find out what program should be used to execute that script. For a Windows-based program, the shebang line might look like this:

```
#!C:\Program Files\SIMPOL\bin\smpcgi32.exe{d}{a}
```

The SIMPOL IDE is especially designed to make building these types of programs quite easy. Simply go into the dialog called from the Project → Settings menu item and add a target. In the target add/modify dialog add the target and the shebang line that is appropriate. When you build the project it will automatically copy the result to the target directory and prepend the shebang line to it. To get it to be called is web server specific. On Apache, in the `httpd.conf` file the line: `Add-Handler cgi-script .smp` would need to be added and if running, Apache would need to be restarted. This program is already available on the Superbase web site: SIMPOL Hello World Sample [http://www.superbase.co.uk/cgi-bin/cgihello.smp].

Other Features

Obviously just being able to respond to requests isn't bad, but there are a number of things that a program needs to be able to do when acting as a web application. One common requirement is to support cookies, both session cookies (that expire when the browser is closed) and standard cookies with an expiration date. The cookie support built into the cgicall type fulfills both of these requirements. Another useful feature is the ability to return content of various types, such as sending a file to be saved on the target machine. This is also supported, since the first line that *must* be sent back by the cgicall object is the `Content-type` line. This is different from the HTML meta tag content-type, since that already presumes a content-type of `text/HTML`. Using this capability it would be possible to create an e-commerce site that sells programs and after payment sends the file automatically to the browser where it can be saved. By not having a download directory that is static, the files are not available except via the program, which can test the user's right to access the download in the first place. It is also possible to interact with web server applications elsewhere that require a content-type that is different from the basic one.

In addition to the content-type and cookie support SIMPOL's CGI implementation also includes full support for both `GET` and `POST`. The correct way to use these items is specific to CGI and is outside the bounds of this document, but there are a multitude of books and web pages that discuss the use of CGI.

Finally, there is also support for retrieving environment variables, form variables, and even the input stream for allowing uploads from browsers directly to the program (such as uploading a company logo as a JPEG). In addition to all of the CGI-specific capabilities, there is still the entire range of capabilities built into SIMPOL. For example, using the CGI support combined with the TCP/IP socket support it is possible to create a web-based email system. Using the tcpsocket type an SMTP email client (even a server) together with a POP3 client could be written. Add to that the support for PPCS and the web pages can even be built based on data in a Superbase database. This would permit any number of web-enabled front-ends to be written to work together with an existing Superbase application. There is simply no advantage to trying to use a product like Microsoft's ASP framework with Superbase database tables via the ODBC driver when access via SIMPOL using PPCS will provide a faster and more reliable solution all from the same software house. Not only that, but once the application has been written and compiled, all that is needed to switch to a Linux or Unix-based web server is to link the appropriate Linux shebang line (which must have only a trailing linefeed) to the front of the already compiled program and place it on the Linux machine in the appropriate location. No change to the source or even the compiled program is necessary!

Web Server Application Tutorial

In this section we will try to build a moderately sophisticated example that uses a design that will allow the program to run using all of the various web server deployment strategies. As our example, we will use the sbisreportfast.sma program provided in the `Projects` directory. The program starts with the function `main()` as shown below.

```
function main(cgicall cgi)
  string sReturnval
  ContactFile cf
  string sISAPIPhysPath

  sISAPIPhysPath = cgi.getvariable("APPL_PHYSICAL_PATH")
  sISAPIPhysPath = .if(sISAPIPhysPath > "", \
                     rtrim(sISAPIPhysPath, "{0}"), "")

  cf =@ init(sISAPIPhysPath)
  sReturnval = fcgi(cgi, cf)
  fcgiterm()
end function sReturnval
```

The interesting thing to note in this function is that there is very little to the function itself. The function calls the `init()` function, then passes the return value from that function to the `fcgi()` function and receives the return value from that and finally calls the `fcgiterm()` before returning the return value from the `fcgi` function. The reason for this design is that although both ISAPI and CGI programs (like almost all SIMPOL programs) begin with the `main()` function, Fast-CGI programs are initialized using the `fcgiinit()` function, subsequent calls only call the `fcgi()` function, and when the Fast-CGI instance is closed, only then will the `fcgiterm()` function be called. To write for all three architectures requires a little bit of planning, so since the Fast-CGI version would never call the `main()` function, everything is designed for the Fast-CGI version and the other two use the `main()` function to call the Fast-CGI components. In the case above, since the `init()` function is used in several places in the contact system, it was decided to have both the `main()` and `fcgiinit()` functions call a common function which in both cases returns what is then required.

Taking a closer look at the beginning of the program another ISAPI-specific item can be seen. That is the request for the variable `APPL_PHYSICAL_PATH`. This variable is only available in ISAPI (and possibly only in Microsoft Internet Information Server (IIS). There are a number of ISAPI-specific variables that can be retrieved, see the IIS documentation for details. The reason that this is so important is that unlike when using CGI or Fast-CGI, ISAPI is done via a Dynamically Linked Library or DLL. DLL's don't have a concept of a current directory when they are executing, so they always inherit the current directory of the parent process, in this case that of the web server. That may or may not be important depending on your web server application, but as you will see later, in this case, knowing the current directory or more importantly the location where the web server application was loaded from is important.

The next thing to note about the `main()` function is the use of a user-defined type called ContactFile. This type was automatically created using an SBL program and in this case is included using the **include** directive and compiled into the program. In other cases, it may be copmiled as a standalone library. It is a type that wraps up a Superbase file that is hosted using PPCS. In the near future a utility program will be created in SIMPOL that produces this from an `SBD` or by interrogating a PPCS-based table. Using this automatically generated type, it is much easier to access the various parts of the `CONTACT` database table used by the sample contact system. Below is the code that makes up the type:

```
//-------------------------------------------------//
//                    CONTACT                      //
//        Constants and Type definitions           //
//-------------------------------------------------//

constant fCONTACTNAME               "CONTACT"
constant CONTACT_LASTNAME           "LastName"
constant CONTACT_FIRSTNAME          "FirstName"
constant CONTACT_CONTACTNO          "ContactNo"
constant CONTACT_PHONE              "Phone"
constant CONTACT_FAX                "Fax"
constant CONTACT_ADDRESS            "Address"
constant CONTACT_CITY               "City"
constant CONTACT_STATE              "State"
constant CONTACT_ZIP                "ZIP"

type ContactFile export
  ppcstype1file  file
  ppcstype1field sLastName
  ppcstype1field sFirstName
  ppcstype1field sContactNo
  ppcstype1field sPhone
  ppcstype1field sFax
  ppcstype1field sAddress
  ppcstype1field sCity
  ppcstype1field sState
  ppcstype1field sZIP
  function open
end type


function ContactFile.open(ContactFile me, ppcstype1 ppcs, \
                          string sIpaddress)
  ppcstype1file f
  integer iErrnum

  iErrnum = 0
  f =@ ppcs.openudpfile(sIpaddress, fCONTACTNAME, error=iErrnum)

  if f !@= .nul
    me.file                    =@ f
    me.sLastName               =@ getfield(f, CONTACT_LASTNAME)
    me.sFirstName              =@ getfield(f, CONTACT_FIRSTNAME)
    me.sContactNo              =@ getfield(f, CONTACT_CONTACTNO)
    me.sPhone                  =@ getfield(f, CONTACT_PHONE)
    me.sFax                    =@ getfield(f, CONTACT_FAX)
    me.sAddress                =@ getfield(f, CONTACT_ADDRESS)
    me.sCity                   =@ getfield(f, CONTACT_CITY)
    me.sState                  =@ getfield(f, CONTACT_STATE)
    me.sZIP                    =@ getfield(f, CONTACT_ZIP)
  end if
end function iErrnum
```

Our next step is to have a look at the `init()` function. It is shown below:

```

function init(string sISAPIPhysPath="")
  ppcstype1 ppcs
  string sIpaddress
  integer iErrnum
  ContactFile cf

  cf =@ ContactFile.new()

  iErrnum = 0
  ppcs =@ ppcstype1.new(udpport=.nul, error=iErrnum, \
                        username="sbiscontact")
  sIpaddress = ""

  getprivateprofilestring(sCGISECTION, sCGIPPCSSERVER, \
                          sDEFIPADDRESS, sIpaddress, \
                          sISAPIPhysPath + sCGIINIFILE, \
                          sCGIINIEOLCHAR)
  if sIpaddress > ""
    if cf.open(ppcs, sIpaddress) != 0
      cf =@ .nul
    end if
  end if
end function cf
```

As we can see from the program code, the primary purpose of this function is to create an object of type ContactFile, create a ppcstype1 object, and then using these two objects and the IP address that is retrieved from a configuration file, to open the `CONTACT` database file. In a complex example this function might be opening dozens of database files for use in a web server application. Earlier we discussed the need to retrieve the physical path to the SIMPOL program in an ISAPI environment. The reason is the code in this function that reads a setting from a configuration file. It would not be a very good design to hard code the IP address and port of the PPCS server, since moving the server would require recompiling the code each time. It is more effective to put these kinds of settings in a configuration file and retrieve them at runtime. The implementation of the function `getprivateprofilestring()` is reasonably compatible with the Windows function of the same name, minus a few limitations and the fact that it works on multiple platforms. It can be found in the `conflib.sml` library in the `lib` directory.

The actual `fcgi()` contains little more than a call to the actual function that does the work, as can be seen below:

```

function fcgi(cgicall cgi, ContactFile cf)
  SBISReportFast(cgi, cf)
end function ""
```

The basic design of a typical web server application uses a sandwich approach. The top of the page is one slice of bread, the bottom of the page is the other, and the output from the program is the filling. If the application is designed carefully making use of cascading style sheets, then changing the look and feel of the web site can be done without even recompiling the program. The way that is done is to use the `HTML_Include()` function to output the top and bottom from files that are located in the directory from where the program is loaded or some other consistent location. This function is part of the `sbislib.sml`.

```
function SBISReportFast(cgicall cgi, ContactFile cf)
  integer iErrnum
  string sTmp, sDateFormat, sTmp2, sTmp3
  ppcstype1record r
  date dt
  objset obsBase
  objsetelementref n
  SBLlocaledateinfo ldiLocale
  string sISAPIPhysPath
  datetime dtStart, dtEnd
  boolean bFound

  sISAPIPhysPath = cgi.getvariable("APPL_PHYSICAL_PATH")
  sISAPIPhysPath = .if(sISAPIPhysPath > "", \
                     rtrim(sISAPIPhysPath, "{0}"), "")

  ldiLocale =@ SBLlocaledateinfo.new()
  sDateFormat = "mmmm dd, yyyy"
  iErrnum = 0
  sTmp = ""
  dtStart =@ datetime.new()
  dtEnd =@ datetime.new()
```

In the initial segment of the `SBISReportFast()` the initialization is done. The function makes use of a number of types and functions provided by libraries that are written and compiled in SIMPOL itself. These types include the objset, objsetelementref, and the SBLlocaledateinfo. The first two types are part of the `objset.sml` library, which provides a set object that operates very similarly to the set object in SBL but which has a key value that must be a string and an optional object reference. This allows the collection and sorting by key value of a set of objects of any type. In this example we will use it for storing the output string in order by a three-level sort. The last of the types is part of the implementation of date format functions to be found in the `SBLDateLib.sml` file. In SBL there are certain global values that determine the formatting for dates, including the names of the days of the week, the months of the year, and the abbreviated months of the year. Since there is nothing global in SIMPOL, this needs to be handled differently. In this program we initialize an object of type SBLlocaledateinfo and then pass it to the functions that require this object. This particular library is compatible with functions found in SBL, so there are no options that would not exist in SBL. There are other libraries being built that provide more sophisticated date formatting routines, though the SBL-compatible ones should be used when working with data from tables via the PPCS type 1 protocol.

The next section of the program checks to see if the return value from the `init()` function actually contains an object or if it failed (returned `.nul`). If it succeeded, it then outputs the content type. Unlike the older Superbase Internet Server product (SBIS), web server programs in SIMPOL can work with any content type desired or required. After outputting the content type and the header, the top of the sandwich is loaded and output by the `HTML_Include` function. Finally, the table is set up and the header is output including the current date.

```
  if cf =@= .nul or cf.file =@= .nul
    CGIFileError(cgi, .nul, "Error opening database \
                             file 'CONTACT'")
  else
    cgi.output("Content-type:  text/HTML{d}{a}{d}{a}", 1)
    cgi.output("<html><head><title>" + sTITLE + \
               "</title>" + CRLF, 1)

    ///////////////////////////////////////
    //         External HTML File        //
    //    Include the header and css     //
    ///////////////////////////////////////
    HTML_Include(cgi, sISAPIPhysPath + "header.htm")

    //////////////////////////////////////
    //          Program title           //
    //////////////////////////////////////
    //cgi.output( CRLF, 1)
    cgi.output('<tr><td><center><h2 class="titledblue">' + \
               sTITLE + '</h2></center></td>'+ CRLF, 1)
    cgi.output('</tr>'+ CRLF, 1)

    /////////////////////////////////
    //      Center the table       //
    /////////////////////////////////
    dt =@ date.new()
    dt.setnow()
    sTmp = DATESTR(dt, sDateFormat, ldiLocale)

    cgi.output('<tr><td align="center"><center><h3 \
                class="titledblue3">' + sTmp + ' - Partial \
                Client Quick Listing</h3></center></td></tr>\
                <tr><td><center><span class="textitalic">Where \
                the first letter of the last name is equal to \
                ''D'' and the result is sorted by <strong>City\
                </strong>, then <strong>LastName</strong>, \
                then <strong>FirstName</strong>.</span>\
                </center><br></td></tr>'+ CRLF, 1)

    cgi.output('<tr><td><center><table border=1 width="510">' + \
               CRLF, 1)
    cgi.output('<tr>' + CRLF, 1)
    cgi.output('<th class="stdhdr" width="25%">City</th>' + \
               CRLF, 1)
    cgi.output('<th class="stdhdr" width="25%">Last name</th>' + \
               CRLF, 1)
    cgi.output('<th class="stdhdr" width="25%">First name</th>' + \
               CRLF, 1)
    cgi.output('<th class="stdhdr" width="25%">Telephone</th>\
                </tr>' + CRLF, 1)
```

Once the preparations are complete, the main part of the program can begin. This is the part of the program that reads the records that match the search criteria, formats the output, and then outputs the result. In this case the program begins by recording the starting time for the search. It then selects the first record in the table according to the LastName index that begins with the letter "D". To make the selection the ContactFile object is used. Each of the properties corresponds to a field in the file with the same name (fields with spaces in the name have the spaces converted to underscores). Also, each field is prepended with a single letter that indicates the data type of the field. Fields that are indexed will have an object associated with their index property and using that the first selection can be made. Afterwards, the record object is used to select the next record in the same index order with which the record itself was selected. As each record is selected and determined to be a valid part of the result set, a string is formulated to hold the three-level sort key, first using the name of the city, then the last name, and finally the first name. Following that another string is assigned the components of the final output, which equates to a row of the HTML table. That string is then added as an object to the objset using the first string as the sort key.

Note

The string is actually passed to the `addelement()` method of the objset by creating a new string using `string.new(sTmp2)`. The reason for this is the objset stores a reference to a string object, not a string itself. If only the `sTmp2` string had been passed, each time it goes around the loop a reference to the exact same string object using a different key would be assigned to the element of the objset so that at the end, all of the elements would point to only one string that contained the last value created. To avoid this, a new string object is created and initialized with the value of the string in `sTmp2`. This ensures that each element references a different string. At that point, the objects are only anchored by the objset, so once the objset goes out of scope, all of the strings are also freed.

This continues until the first non-matching record is found in the index. At that point, the loop exits and the objset contains all of the results in the desired sort order.

```
    // Record the starting time for the search
    dtStart.setnow()

    // Select the first record that starts with a D in the
    // Lastname index and then continue to select records
    // until the first letter is no longer a D.
    bFound = .false
    r =@ cf.sLastName.index.selectkey("D", error=iErrnum, \
                                      found=bFound)

    obsBase =@ objset.new()

    //SELECT ;
    //WHERE Lastname.CONTACT LIKE "D*"
    //ORDER City.CONTACT,LastName.CONTACT,FirstName.CONTACT
    //TO APPEND
    //END SELECT

    // This section performs the actual report and stores
    // the sort key plus the desired output into the set.
    // The set will automatically be stored in sorted order.

    while r !@= .nul and .lcase(.lstr(r.get(cf.sLastName),\
                                      1)) == "d"
      sTmp = PAD(r.get(cf.sCity), 40) + \
             PAD(r.get(cf.sLastName), 60) + \
             r.get(cf.sFirstName)

      sTmp2 = '<tr><td class="stdtext">' + \
              .lstr(r.get(cf.sCity),15) + \
              '</td><td class="stdtext"><a href="' + \
              HTML_Page("sbiscontactdisplay.smp", cgi) + \
              '?cno=' + r.get(cf.sContactNo) + '">' + \
              .lstr(r.get(cf.sLastName),15) + \
              '</a></td><td class="stdtext">' + \
              .lstr(r.get(cf.sFirstName),15) + \
              '</td><td class="stdtext">' + \
              .lstr(r.get(cf.sPhone),10) + '</td></tr>' + CRLF

      obsBase.addelement(sTmp, string.new(sTmp2))
      r =@ r.select(.false, error=iErrnum)
    end while iErrnum > 0
```

Finishing the report is now just a matter of retrieving the first element of the objset, outputting the element, and then retrieving the next element until we run out of elements. There is no real need to find out how many elements there are, since we can just continue until either the returned element is equal to a reference to `.nul` or the `t` property is equal to a reference to `.nul`. Then we retrieve the time again and output the rest of the the table and close up the remaining bits of the HTML.

```

    // This part now outputs the results of the report that
    // had been stored in the set while gathering the
    // results. The output comes out in the correct order
    // sorted three levels deep, by using a combined key
    // composed of the City, the lastname and the first name
    // where each of the first two have been padded to 60
    // characters wide.
    n =@ obsBase.getfirst()
    while n !@= .nul and n.t !@= .nul
      cgi.output(n.t.element, 1)
      n =@ n.t.getnext()
    end while

    dtEnd.setnow()

    // After report section
    cgi.output('</table>' + CRLF, 1)
    cgi.output('<p><strong>Total: ' + \
               .tostr(obsBase.totalcount, 10) + " match" + \
               .if(obsBase.totalcount <> 1,"es","") + \
               ' found from a total of ' + \
               .tostr(cf.file.recordcount(error=iErrnum), 10)\
               + ' records. The total search time was ' + \
               .tostr((dtEnd - dtStart)/1000000, 10) + \
               ' seconds.</strong>\
               </p><br></center></td></tr>' + CRLF, 1)
    ////////////////////////

    ///////////////////////////////////////
    //    Include external HTML file     //
    //           as the footer           //
    ///////////////////////////////////////
    HTML_Include(cgi, sISAPIPhysPath + "footer.htm")
```

```
  end if
end function ""
```

The very end of the program occurs when the `fcgiterm()` is called either from the `main()` function or directly by the Fast-CGI support. In this case, there is nothing for the function to do, so it is empty.

CGI Samples

As mentioned earlier, on the Superbase web site their are a number of samples of programs that are already running using SIMPOL to serve dynamic web pages. More will be added as time passes. To access the samples visit the page: SIMPOL Samples Page [http://www.superbase.co.uk/simpolsamples.htm]. On this page there are links and explanations as well as the ability to view the source code of each program.

Part IV. Using Databases

Working with databases is an important part of most programming languages that deliver dynamic web content or that are used for desktop application development. SIMPOL comes ready to work with different types of database content but the one thing that they have in common is that they are all represented using objects. In this part we will learn how to access databases in Superbase PPCS format and also use the new SBME format for single program multithreaded access. Eventually there will also be objects in SIMPOL for using volatile databases (typically hosted only in memory) and for accessing various SQL databases, but they will come later.

Table of Contents

Chapter 12. Using Databases in SIMPOL

This chapter will describe the approach to databases using objects in SIMPOL. This chapter should be read first before trying to use any of the database access technologies from SIMPOL.

Terminology

A good place to start in this chapter is a discussion of terminology. In traditional desktop databases like Superbase, dBase, FoxPro, and Paradox, it is common to refer to database files, fields, indexes, and records. In SQL databases it is more common to use the terms tables, columns, and rows. Loosely it would be accurate to say that database files equate to tables, fields equate to columns, and rows equate to records. In SIMPOL we tend to stick with the desktop database terminology when refering to non-SQL data sources, although we are attempting to standardize on the terms: tables, fields, and records. The reason for selecting tables rather than files is that in some databases more than one table can be stored in the same physical file, as is the case with Microsoft Access and with the new Superbase Micro Engine. That would result in overuse of the word file and as such the decision was made to use table instead.

Traditional File-Oriented Databases

Introduction

In traditional desktop database environments such as Superbase and dBase, there are numerous commands in the programming languages to handle the various tasks associated with working with database files. In some cases the command set can be quite large and often somewhat ambiguous in that command names may be reused in various combinations with slightly or even widely different effects. The result is that generally the programmer is required to remember a large number of commands with various parameters in order to accomplish very basic manipulation of the database. Since in most cases these commands are considered key words in the language, they also reduce the available group of obvious variable names that can be used by the programmer.

SBL Database Commands

SBL has a large array of database oriented commands that are composed of one or more key words in the programming language. For example, in Superbase Basic Language (SBL) there is a wide variety of `SELECT` commands. These include:

- `SELECT FIRST`
- `SELECT LAST`
- `SELECT NEXT`
- `SELECT PREVIOUS`
- `SELECT CURRENT`
- `SELECT KEY`
- `SELECT DUPLICATE`
- `SELECT WHERE`
- `SELECT REMOVE`
- `SELECT`

Each of these commands also has various parameters that can be appended to the end or in some cases inserted earlier in the command. There are numerous other commands that exist purely to manipulate databases.

Common Database Programming Problems

One of the biggest issues by far, however, which people run into when working with these languages for manipulating databases is what I call the *current everything* problem. The assumption is made that files are globally visible and that field names in files are globally visible identifiers. Although that simplifies things when doing simple things, it results in great complexity and confusion when doing more complex things. It also results in limitations such as not being able to open the same file twice in the same instance of the program because there would be no way to differentiate between the two versions (or only with great complexity).

A common error made by SBL programmers is that of selecting a record using an index different to the *current* index as seen from the user's perspective, and then after establishing that the selection worked, reselecting the record with a lock. Unfortunately, the second selection occurs using the current index, which is not that used when they selected the record the first time and so they lock the wrong record and possibly make and save changes to the wrong record. See the code in the example below:

```
' The current index is LastName and the current file is ADRB
SELECT KEY 12345 FILE "ADRB" INDEX RecNo.ADRB
IF FOUND ("ADRB") THEN
  ' Here the SELECT CURRENT LOCK operates on the LastName
  ' index which is whatever was current before the
  ' SELECT KEY took place against the RecNo index.

  SELECT CURRENT LOCK

  ' Correct would have been to use:
  ' SELECT CURRENT LOCK FILE "ADRB" INDEX RecNo.ADRB
  ' but this is a common error.

  AccountBalance.ADRB = AccountBalance.ADRB + deposit%

  ' The program now assigns the deposit amount into the wrong
  ' account number and stores it. This problem will be difficult
  ' to track down because the deposit to the wrong account won't
  ' always happen. If the current index is RecNo, then the code
  ' will work, if it is not, then the deposit will be made to
  ' whichever record is current in the current index.

  STORE
END IF
```

In an object-oriented environment these kinds of issues don't exist since all selections are made as a call to a method of an object, so there is no way that you can accidentally call the wrong object (at least not easily).

Object-Oriented Database Access in SIMPOL

Introduction

There are a number of different object-oriented approaches to working with databases and in general those methods are directly related to the type of database access that they attempt to model. With SQL style access the approach tends toward recordset objects, which is completely understandable since SQL is all about doing queries and doesn't really have an idea of direct table access. That is reserved to the routines that actually implement a SQL database engine. In SIMPOL the database access is based around accessing database sources (which can be files or servers) and then from that file or server accessing tables. Each table has a ring of field objects and a ring of index objects. Record objects are created as the result of selecting a record using one of the selection methods. Index objects carry a reference to the field for which they are an index. In later releases there may be a ring of objects that describe the elements of an index that is either multiple field or based on something other than field information.

One big difference between the older Superbase approach to working with tables and field names and the object-oriented version is that opening a table does not simply create a group of globally visible identifiers. This means there is a little more work involved when using objects. At the same time, there is no limit to how many times a table can be opened concurrently or how many records could be the *current* record. Since records are objects whenever a record object exists it is available.

Another significant difference is in the way that records are selected. The object-oriented approach takes a little getting used to, but is perfectly logical and will eventually feel quite natural and obvious. Table objects can either select the first or last record in a table in sequential order. Index objects can select either the first or last record in index order or select via a key value into the index. So what about selecting the next or previous record? That is reserved to record objects. Only a record object knows how it was selected and therefore its index position. A record can also select only the next or previous record according to the order that was used to select the record, either in index order or sequential order of the table. To change the index of a record the record can be reselected with an option to change the index.

Database Type Tags for Generic Database Functionality

In keeping with the object-oriented design of SIMPOL a set of type tags (see the section called “Value Types, Reference Types, and Type Tags”) was created for use with databases that are of a consistent form. This set of type tags is known as the *db1* set. This set of tags includes the following:

- `db1table`
- `db1field`
- `db1index`
- `db1record`

By writing generic functions to use the type tags rather than the specific object type declarations it makes it very easy to switch between different database types without rewriting the functionality for each individual type.

Note

One difference between sbme1 types and ppcstype1 types is that the former have a table or tablename property where the latter have a `file` or `filename` property in addition to the tablename. This does not affect the use since when working with the `db1*` type tags the table can be used in most cases. A more signficant difference is that ppcstype1fields have a much greater array of properties than sbme1fields. They include help text, comments, display formats, and other things that are not provided by the more storage-oriented and lower level engine from sbme1. In general, generic database routines should not be dependent on a display format that is provided with a column (or column widths that may not exist, etc.).

A Comparison of SBL Commands and SIMPOL Methods

Comparing the two approaches should help to clarify much of the difference in approach between the command based and the object based methods. To summarize and compare the SBL commands to the SIMPOL object methods then, here is a small table:

Table 12.1. Comparison of SBL file access commands to SIMPOL methods

SBL	SIMPOL
`SELECT FIRST INDEX ""`	`db1tablevar.select(lastrecord=.false)`
`SELECT LAST INDEX ""`	`db1tablevar.select(lastrecord=.true)`
`SELECT FIRST INDEX RecNo.TEST`	`db1indexvar.select(lastrecord=.false)`
`SELECT LAST INDEX RecNo.TEST`	`db1indexvar.select(lastrecord=.true)`
`SELECT KEY 123 INDEX RecNo.TEST`	`db1indexvar.selectkey(123)`
`SELECT NEXT`	`db1recvar.select(previousrecord=.false)`
`SELECT PREVIOUS`	`db1recvar.select(previousrecord=.true)`
`SET INDEX Name.TEST`	`db1recvar.selectcurrent(db1indexvar_Name)`
`SELECT FIRST LOCK INDEX ""`	`db1tablevar.select(lastrecord=.false, lock=.true)`
`SELECT FIRST LOCK INDEX RecNo.TEST`	`db1indexvar.select(lastrecord=.false, lock=.true)`
`SELECT KEY 123 LOCK INDEX RecNo.TEST`	`db1indexvar.selectkey(123, lock=.true)`
`SELECT NEXT LOCK`	`db1recvar.select(previousrecord=.false, lock=.true)`
`SELECT CURRENT LOCK`	`db1recvar.selectcurrent(lock=.true)`
`SELECT REMOVE`	`db1recvar.delete()`

There are numerous other combinations, but the previous table should show a reasonable cross-section. One of the interesting abilities in SIMPOL is that of deleting a record but still having the record available. When the `delete()` method is called, the record is deleted but the record object still exists. Its stored property is set to `.false` and it is reset internally to appear like a new record that has been filled in. That means that the record could now be saved as a new record (possibly with some modification).

Summary

In this chapter we have looked at the generic differences between working with databases in command-oriented languages like SBL and the methods employed by SIMPOL. In the following chapters we will discuss the specific issues affecting access to PPCS and SBME databases in SIMPOL.

Chapter 13. Using PPCS in SIMPOL

This chapter will describe in detail the issues surrounding database access in SIMPOL using the Superbase Peer-to-Peer Client/Server (PPCS) approach. By the end of the chapter you should feel reasonably confident in accessing PPCS database tables from SIMPOL.

What is PPCS?

PPCS provides a protocol for accessing database tables and binary files using a variety of communication methods. Currently there is support for direct serial cable connections using RS-232 and also modem-based connectivity. There is also support for NetBIOS and UDP/IP. PPCS is a connectionless protocol. In practical terms, that means that there is no actual maintained connection between the client and the server. Each transaction between the client and the server is complete and independent of any other connection. For example, assume that a UDP connection is made across the Internet and a database table is opened and the client selects a record via an index. Then for some reason the connection to the Internet goes down. After a short while the connection comes back up and the user selects the next record. In such a situation, since no request had been made to the server in the interim, the PPCS client program would not be aware that anything had happened and would successfully receive the next record. This would not be the case with a connected protocol.

As described in the previous paragraph, PPCS allows access to database tables and binary files that have been shared on a PPCS server. The type of access provided is record level access to the database tables in a shared read-write mode. That means that only operations that can work in a shared read-write mode are supported. Records can be created, locked, modified and deleted using this access technology. Not supported is changes to the database structure, such as adding, modifying, or removing fields and indexes or even creating or removing database tables. In SIMPOL this is known as PPCS Type 1. At some stage we will release a PPCS Type 2. This will support the full range of capabilities of the new Superbase Micro Engine database and will also allow for complete remote management of the database backend. The binary file support allows the sharing of binary files for direct transfer via PPCS from the server to the client. The name space within a PPCS Type 1 server is flat, so binary and database files with duplicate names even if from different directories cannot be served on the same server.

In SIMPOL there is currently only support for using PPCS via UDP. Preparation has been made to support serial connections, but the actual capability has not yet been implemented. Also, it is not possible to transfer binary files using PPCS in SIMPOL. By using the TCP/IP sockets a file transfer protocol can easily be created. Sample programs including a client and server program are provided.

Setting Up a PPCS Server Using Superbase

Setting up a PPCS server using Superbase is relatively easy, especially if you are using Superbase 2001 or later. At the very least, Superbase 3.6i build 478 or later is required since that is the first release of Superbase with the PPCS technology. In all cases there is a remote connections wizard provided with each version and a document included in Adobe Acrobat format that describes how to set up a PPCS server, client, etc. using the wizard. If you are using 2001 or later, the wizard can save off the configuration as a standalone program for later use (and modification).

Note

SIMPOL Professional includes a SIMPOL-based PPCS server engine and is licensed for three concurrent users for testing purposes. The server engine and a `readme.txt` file that explains how to use it can be found in the `simpolserver` directory in your SIMPOL installation.

Object-Oriented Database Access

Unlike the standard command-based approach provided with SBL, dBase, and other products, in SIMPOL database access is done via objects. To use the database access in SIMPOL easily and effectively it is very useful to learn about the logic behind the decisions on how the database objects were designed. Once the underlying system is clear, it will feel very natural using the database objects to accomplish database-oriented tasks in SIMPOL. Also, the object-oriented approach will completely eliminate many sources of errors that occurred in SBL programs in the past. As is true in general in SIMPOL, there is nothing global about database objects. There is no current record, current file, current index, etc. Instead, it is possible to have as many current records as required, simply by having a different variable (or element in a set, or element in an array) that references each record object.

In the beginning, there is the ppcstype1 object. This is used to open tables from various backend PPCS servers. Unlike the implementation in SBL, it is only necessary to create one of these per communication method, since the connector is not specific to a target backend (in the case of UDP connections — in the case of serial connections it would be necessary since this is a hardware limitation). Using the ppcstype1 object, a database table is opened from a backend and a reference to the database file is assigned to a ppcstype1file variable. At this point we can already retrieve records in sequential order from the database table, either starting at the beginning or the end. The database file object holds all of the information necessary to analyze the structure of the database table. There is a firstfield property that holds a reference to the first field in the database table definition. It contains a property called next that holds a reference to the next database field in the file definition. The next property of the last field in the file definition will contain a reference to the first field in the definition thus forming a ring. A similar ring exists for the indexes.

When a record is selected from the database using either an index object, a database table object, or a record object, then a ppcstype1record object is returned. This object contains the data, but has no concept of the actual file description, fields, etc. The reason for this is fairly technical, but essentially for it to do so, it would have needed to be a record object for a specific data type created at the time the database was opened that was designed specifically for that database file and would have made it impossible to easily use the same variable for records from different database tables. As such, to access the data for a given field from a record object it is necessary to make use of the ppcstype1field object for the field from which the information is to be read (or to which the data is to be assigned). There is a `get()` method and a `set()` method for reading and writing data from and to the various fields of a record. Both take a field reference as a parameter although the `set()` method also takes a value as its second parameter. Another approach is to use the member operator. This was specially implemented for accessing the data in the record object in a visually more elegant way, but has several disadvantages, such as the fact that it can't use a variable, and if a field name is passed that is incorrect (including incorrect case) then it will cause a runtime error that cannot currently be trapped. The example below will demonstrate accessing a record from a table via PPCS and reading the values from the fields.

The beginning of the program starts as usual in the function `main()`. We begin by declaring the various variables that will be needed for this function. The remainder of the description can be found directly in the comments of the program itself.

```
function main()
  string sResult
  integer iErrnum
  ppcstype1 ppcs
  ppcstype1file f
  ppcstype1record r
  ppcstype1field fld
  ppcstype1index idx
```

```
boolean bFound

// iErrnum MUST be initialized or there will be
// no object in which to return the result
iErrnum = 0

// We now attempt to create a ppcstype1 object which should
// almost always work. In this case we pass .nul to the
// udpport parameter because we don't care which port we
// get, we just want one. We have to pass .nul to the named
// parameter udpport since otherwise the function won't know
// whether we want a UDP or a serial (not yet supported)
// connection.
ppcs =@ ppcstype1.new(udpport=.nul, error=iErrnum, \
                      username="example1")

// We can test for an error value here or for the ppcs
// variable containing .nul
if ppcs =@= .nul
  sResult = "Error number " + .tostr(iErrnum, 10) + \
            " creating ppcs object!{d}{a}"
else
  // Now we are going to open the database table using
  // the ppcstype1 object
  f =@ ppcs.openudpfile("ppcs.superbase.co.uk:1280", \
                        "CUST", error=iErrnum)
  if f =@= .nul
    sResult = "Error number " + .tostr(iErrnum, 10) + \
              " opening file 'CUST'!{d}{a}"
  else
    // We got this so far so we have the table open

    // Now we are going to locate the reference to the
    // index we want to use for selecting the record.
    // The following loop will start at the firstindex
    // and then go around until it either finds the desired
    // index or returns to the first index in the ring.
    bFound = .false
    idx =@ f.firstindex
    while idx !@= .nul
      if idx.field.name == "LastFirstName"
        bFound = .true
      else
        idx =@ idx.next
      end if
    end while idx =@= f.firstindex or bFound

    if not bFound
      sResult = "Index 'LastFirstName' not found!{d}{a}"
    else
      // We found the index for our test so now we select
      // the record that we are looking for using the
      // selectkey() method.
      r =@ idx.selectkey("Johnson, Amanda", error=iErrnum)
      // If the selection fails then r will not refer to an
      // object because we did not pass a found parameter to
      // the method. When using an inexact match that we
      // expect might fail (like looking for Joh*) we would
      // pass a found parameter so that we always get the
      // closest matching record returned.
      if r =@= .nul
        sResult = "Error number " + .tostr(iErrnum, 10) + \
                  " retrieving record!{d}{a}"
      else
        // We found the record we were looking for, so let
        // us now output the contents of the record. This
        // time we will use the get() method and a loop that
        // tests the datatype of the field to allow us to
        // format it properly. We loop around the fields in
        // the file retrieving each field's name and value
        // and then add it to the result string.
        fld =@ f.firstfield
        sResult = "Record for Amanda Johnson:{d}{a}"
        while
          if fld.datatype =@= string
            sResult = sResult + "  " + fld.name + ": " + \
                      r.get(fld) + "{d}{a}"
          else if fld.datatype =@= integer or \
                  fld.datatype =@= date or \
                  fld.datatype =@= time
            sResult = sResult + "  " + fld.name + ": " + \
                      .tostr(r.get(fld), 10) + "{d}{a}"
          else if fld.datatype =@= number
            sResult = sResult + "  " + fld.name + ": " + \
                      .tostr(.fix(r.get(fld), 100), 10) + \
                      "{d}{a}"
          end if
          fld =@ fld.next
        end while fld =@= f.firstfield

        // As you can see from the code above, we treat
        // dates and times as if they were integers. This
        // is because there are no built in functions to
        // format a date, a time, or a datetime. The reason
        // is that there are too many different ways this
        // might be done for different locales and it is
        // best solved with a SIMPOL-based library. Such a
        // library is part of the current distribution but
        // not relevant for this example.

        // Now let's get the next record in the same index
        // and output that.
        r =@ r.select(error=iErrnum)
        if r =@= .nul
          sResult = sResult + "Error number "  + \
                    .tostr(iErrnum, 10) + \
                    " retrieving record!{d}{a}"
          else
            // Again we succeeded in getting the next record
            // without error, so this time we will output the
            // fields expressly using the member operator for
            // the ppcstype1record object. The advantage to
            // using the member operator is the code looks
            // easier to understand. The disadvantage is that
            // if the field name changes the code will break
            // whereas the previous version would not. The
            // previous version neither knows nor cares what
            // the fields are called.
            sResult = sResult + "{d}{a}Next Record:{d}{a}"
            sResult = sResult + "  RecNum: " + \
                      .tostr(r!RecNum, 10) + "{d}{a}"
            sResult = sResult + "  Firstname: " + \
                      r!Firstname + "{d}{a}"
            sResult = sResult + "  Lastname: " + \
                      r!Lastname + "{d}{a}"
            sResult = sResult + "  Organization: " + \
                      r!Organization + "{d}{a}"
            sResult = sResult + "  Street: " + \
                      r!Street + "{d}{a}"
            sResult = sResult + "  City: " + r!City + "{d}{a}"
            sResult = sResult + "  Country: " + \
                      r!Country + "{d}{a}"
            sResult = sResult + "  LastFirstName: " + \
                      r!LastFirstName + "{d}{a}"
            sResult = sResult + "  CreditLimit: " + \
                      .tostr(.fix(r!CreditLimit, 100), 10) + \
                      "{d}{a}"
            sResult = sResult + "  Balance: " + \
                      .tostr(.fix(r!Balance, 100), 10) + \
                      "{d}{a}"
          end if
        end if
      end if
    end if
  end if
end function sResult
```

As we can see from the previous program, there is a little bit more overhead when accessing the parts of the database programmatically from SIMPOL as compared with SBL, but there is absolutely no possibility of errors in the SIMPOL method, since the record object is always a known quantity. There are also a larger number of ways to write things so that the program code does not need to know too much about the actual data to still do its job. This allows us to write more generic and library code that gradually adds to our ability to do a job more quickly.

Almost every bit of the preceding program could be applied to accessing a database using the sbme1, simply by changing the data types and the opening method. In SIMPOL a great deal of effort has been invested to ensure that program code will be able to deal with different database types without needing to be greatly rewritten.

Chapter 14. Using SBME Databases in SIMPOL

This chapter will describe in detail the issues surrounding database access in SIMPOL using the Superbase Micro Engine (SBME). The SBME is a new, next-generation database design that incorporates support for all of the value data types (as well as some additional data types that have a single value for an object of that type that is of one of the value types) that are included in the SIMPOL programming language. The database engine has few limitations and is extremely fast with a very small footprint (ca. 150 KB).

Note

The single-user engine is accessed using the "sbme" component. The multiuser engine currently only provides PPCS Type 1 access. This engine is included as a three-user test version on the same machine where the IDE is installed. The multi-user engine component "ppsr" is the only current way to access the new database engine for multi-user access. A later multi-user engine will provide PPCS Type 2, which will allow for the full array of capabilities provided by SIMPOL. If you are planning to use the multi-user engine then it is recommended that no use be made of blob, boolean, or datetime fields since these are not supported for mapping to PPCS Type 1.

Introduction

The SBME database engine is a high-performance database engine that provides a fairly low-level API for accessing database tables and records. It is not an SQL-style API but rather a table and record-oriented one. The current engine provides a storage-only database (no calculations, constants, validations, triggers, etc.). The format is as follows:

- SBME database files have an `sbm` extension
- The file can contain one or more database tables
- All of the parts of the database table are contained within the database file
- Each table consists of one or more fields and 0 or more indexes
- A field has a datatype that must be one of the value types or else a date, time, or datetime
- An index is currently associated with a specific field, though in future may be over multiple fields

One of the more significant points to be aware of is that there is no column width or display format associated with a field.

Difference Between SBME and SBF's

There are a number of differences between the SBME database design and that of the older Superbase format. One thing that immediately is noticeable is the lack of a column width or display format. Another big difference is that there are no field characteristics like "read only", "required", "non-stored", nor any validation, default or calculation formulae. The reasons for this are numerous. Ovedr the years most of the more advanced Superbase programmers have found that the use of these features tends to cause more trouble than they are worth. The only ones to profit from the use of such features tend to be very simple databases with little or no significant program code. The more complex an application becomes the more restrictive the use of these features becomes. As such, the right place to put these types of features is in the code that is responsible for saving records in a given table.

One of the common complaints voiced by some customers was the fact that Superbase databases tend to clutter up a directory with many files (`sbd`, `sbf`, `sb!`, `1 - 999` — indexes). Other customers liked the ability to change a file definition by simply copying over the `sbd` file (though this is not generally recommended). In the new design, all of the components of the database file are in one container. Optionally more than one database table can be stored in the same container. Because the SBME API is so low-level, there is no built-in referential integrity, data dictionary, etc. but if desired, much of this can be implemented inside of any `sbm` file.

Programming with SBME Databases

Essentially, working with SBME databases from a programmatic standpoint is virtually no different to working with PPCS databases. For a fairly in-depth description please see the section called "Object-Oriented Database Access" and Chapter 12, *Using Databases in SIMPOL*. The only significant differences arise in the creation and opening of SBME databases. To open an SBME database, the `new()` method of the sbme1 is called. In that call the name of the database file and the action to take must be specified. The action can be one of the following characters or combinations of characters:

- O — open
- C — create
- R — replace
- OC — open if exists otherwise create
- RC — replace if exists otherwise create

Once the file has been opened the sbme1 object can be used to retrieve the list of tables and if the table name is known the `opentable()` method can be used to retrieve a reference to the table. From that point onwards things work identically to the way PPCS works for accessing fields, indexes, and records other than a difference in terms of the types of information available for the actual objects (properties and methods for things like readonly, required, etc.).

Creating an SBME database is done by first opening or creating an `sbm` file and then by calling the `newtable()` method. The following function is part of the `db1util.sml` library that is included in the `lib` directory. The entire source code to the project can be found in the `projects\libs\db1util` directory.

```
function create_sbme1table_from_db1table(type(db1table) dbSrc, \
                                          sbme1 sbmFile,
                                          string sNewTableName) export
  integer iResult, iErrnum
  string sTablename
  type(db1field) fld
  sbme1newtable sbmnt
  sbme1newfield sbmnfld
  sbme1newindex sbmnidx

  iResult = iIMEX_ERR_SUCCESS
  if dbSrc =@= .nul
    iResult = iIMEX_ERR_PPCSOBJNUL
  else if sbmFile =@= .nul
    iResult = iIMEX_ERR_SBMEOBJNUL
  else
    if sNewTableName > ""
      sTablename = sNewTableName
    else
      iErrnum = 0
      if dbSrc.type =@= ppcstype1file
        sTablename = dbSrc.filename
      else if dbSrc.type =@= sbme1table
        sTablename = dbSrc.tablename
      end if
    end if

    if sTablename > ""
      iErrnum = 0
      sbmFile.lock("shared", iErrnum)
      if iErrnum != 0
        iResult = iIMEX_ERR_SBMEOBJLOCKED
      else
        sbmnt =@ sbmFile.newtable(sTablename)
        if sbmnt =@= .nul
          iResult = iIMEX_ERR_SBMETABLECREATEFAILED
        else
          fld =@ dbSrc.firstfield

          while
            sbmnfld =@ sbmnt.newfield(fld.name, fld.datatype)
            if fld.index !@= .nul and fld.datatype !@= number
              sbmnidx =@ sbmnt.newindex(sbmnfld, 100, "")
            end if

            fld =@ fld.next
          end while fld =@= dbSrc.firstfield

          sbmnt.create(iErrnum)
          if iErrnum != iSIMPOL_ERR_SUCCESS
            iResult = iIMEX_ERR_SBMETABLECOMMITFAILED
          end if
          sbmFile.commit()
        end if
        iErrnum = 0
        sbmFile.unlock(iErrnum)
      end if
    end if
  end if
end function iResult
```

From the preceding program code, specifically following the statement `if sTablename > ""` the order of events when creating a new table is:

- Get at least a shared lock on the sbme1 object using the `lock()` method

- Create a new table with the desired name using the `newtable()` method, which returns a sbme1newtable object
- For each field desired create a new field using the `newfield()` method of the sbme1newtable object
- If a field should also be indexed then create an index on the new field using the `newindex()` method of the sbme1newtable object and passing the reference to the sbme1newfield object
- Create the table by calling the `create()` method of the sbme1newtable object
- Call the `commit()` method of the sbme1 object to write the changes back to the file
- Call the `unlock()` method of the sbme1 object

There is also a number of tools and library modules being created that are intended to make importing data and creating files easier. Watch the `projects` directory for ongoing changes.

Part V. Calling SIMPOL Functions as DLL Calls

We felt that one way to make SIMPOL available would be to allow the calling of SIMPOL functions as if they were DLL calls. This could be useful for interacting with other Windows programs and would also assist existing Superbase users to gradually port their applications to the new language by allowing them to write new functionality in SIMPOL but still call the functions from the older Superbase language. In this part of the reference guide we will explore how to call SIMPOL functions and programs from Superbase (and other languages) as if they were DLLs.

Table of Contents

Chapter 15. Calling SIMPOL Functions as DLL Calls

Introduction

In this chapter we will discuss how we might call a SIMPOL program or even just a function as a DLL call from Superbase or Visual Basic. In order to use this functionality from Superbase it is essential that the user be running Superbase 3.6i build 496 or later, since only as of that release is it possible to call Win32 DLLs.

This functionality is not in as flexible a form as may be provided over the long term, but it is provided in a usable way in order to promote interoperability between existing Superbase applications and SIMPOL programs. There are a few limitations such as only a single string argument can be passed to the function being called in SIMPOL and only a single string can be returned as the result of the function call to the calling program. This is partly related to the differences in datatypes that are available to each language. In SIMPOL there are virtually no limitations on the size and precision of numeric types and the language supports both `.nul` and `.inf` as special values. Strings are also based on Unicode and are essentially unlimited in size. At another point in the future, there will be a method of accessing SIMPOL functions as DLL calls for interoperability but then it will be specifically designed to work with Win32 (and Linux and Mac OS-X) in as transparent a manner as possible.

In spite of the fact that only one string can be passed as an argument to the SIMPOL function, if it contains **TAB**-delimited items these will be asigned to separate string parameters in the target function.

Using SMEXEC

There are three API functions that are provided in SMEXEC for allowing calls to SIMPOL functions from external applications and languages. Those three functions are:

- `SMExec_LoadSMPModule()`
- `SMExec_UnloadSMPModule()`
- `SMExec_RunSMPFunction()`

The first of these functions is used to load a compiled SIMPOL program (either an SMP or an SML). The second is used to free that program and release the memory that it is using. The third is used to call a function in the program that was loaded. It is important that the return value from the load function be tested for success (0). If the load has failed, then none of the other functions should be called. Calling the unload function with an invalid handle can also result in a general protection fault (GPF).

SMEXEC Example Using SBL

In this section we will examine a source code program in SBL that makes calls to a SIMPOL library called `jpeglib.sml`. The complete SBL source code for this program is called `SMEXEC.SBP` and is included in the `samples/SBL` directory.

The program below begins by declaring a few variables that simply hold the location of the various paths for parts of SIMPOL. It also stores the current directory so that it can be restored later and prepares Superbase for using API calls.

Note

The following program is written in the Superbase Basic Language (SBL). To fit it into the available space, lines have occasionally need to be continued on the following line. The SIMPOL line continuation character, the backslash, has been used for this. However, SBL does not have a line continuation character, so wherever this character is used the following line must be rejoined to the line containing the character and the character must be removed!

Example 15.1. SBL program calling SIMPOL function

```
SUB main()
   DIM cd$,simpolpath$,simpolbin$,simpollib$

   simpolpath$ = FN spath("C:\Program Files\SIMPOL\")
   simpolbin$ = simpolpath$ + "BIN\"
   simpollib$ = simpolpath$ + "LIB\"

   cd$ = DIRECTORY
   REGISTER CLEAR
```

The next line of the program does a change of directory to the SIMPOL `bin` directory. This is necessary because the `SMEXEC32.DLL` is statically linked to the `SMPOL32.DLL` and if it is not in the current directory then the operating system won't be able to find it unless it is added to the path. SIMPOL is not installed such that anything has to be added to the path and therefore it is better to make this change. In the future the change should not be necessary, but to use the functionality at the time of writing, it is necessary to be in the location of the DLLs.

```
   DIRECTORY simpolbin$
```

It is also necessary to use the fully qualified path name when laoding the DLL on Windows NT, 2000, and XP because there is no guarantee that on those OS's that the operating system will look for the DLL in the current directory. The format used in this example will work on all operating systems and is therefore recommended. If you put the SIMPOL `bin` directory into the path, then the full path names are not required. In the standard installation the directory is not added to the path.

```
   ' * SBUINT UTILFUNC
   '   SMExec_LoadSMPModule(PSBUBYTE pmodname,
   ' *                      SBRAMSIZE modnamecharcount,
   ' *                      SBRAMSIZE bytesperchar,
   ' *                      PPVOID ppmodinfo)
   REGISTER simpolbin$ + "SMEXEC32.DLL",\
            "SMExec_LoadSMPModule","JCJJM"

   ' * SBUINT UTILFUNC SMExec_UnloadSMPModule(PVOID pmodinfo)
   REGISTER simpolbin$ + "SMEXEC32.DLL",\
            "SMExec_UnloadSMPModule","JJ"

   ' * SBUINT UTILFUNC
```

```
' * SMExec_RunSMPFunction(PVOID pmodinfo,
' *                       PSBUBYTE pfuncname,
' *                       SBRAMSIZE funcnamecharcount,
' *                       PSBUBYTE pparams,
' *                       SBRAMSIZE paramcharcount,
' *                       PSBUBYTE poutput,
' *                       SBRAMSIZE maxoutputcharcount,
' *                       PSBRAMSIZE poutputcharcount,
' *                       SBRAMSIZE bytesperchar )
REGISTER simpolbin$ + "SMEXEC32.DLL",\
         "SMExec_RunSMPFunction","JJCJCJFJMJ"
```

The program now declares a few variables that are used with the calls to the SMEXEC functions. It then clears the screen, prints the start time (this for doing time tests of calls to SIMPOL functions), and assigns the SIMPOL program file and function names to their respective variables.

```


DIM smp$,func$,params$,h&%(10),res$,pos%%

CLS

? "Start time: " + TIME$ ( NOW ,"hh:mm:ss.sss") + " - ";
? DATE$ ( TODAY ,"dd mmm yyyy")
?
smp$ = simpollib$ + "JPEGLIB.SML"
func$ = "smexec_getjpegsize"
```

Now we load the compiled SIMPOL program/library (`*.smp` or `*.sml`). The parameters to the function call are the program/library name (`smp$`), the length of the program/library name parameter (`LEN (smp$)`), the number of bytes per character (always `1` in SBL but it could be `2` if called by a program that is operating internally in Unicode), and a pointer or reference to a long integer where the handle can be stored if the function is successful (`h&%(1)`). In our example, if the call to load the library is successful, a handle to the library is returned in the variable `h&%(1)`. It is necessary to use an array here since the function needs to assign the handle and this is the only reasonable way to do that in SBL. Array variables are passed by reference rather than by value. It is also possible to load any number of SIMPOL programs/libraries at the same time and to make calls to them as needed, freeing them at the end. They do not need to be loaded and then unloaded right away.

```


h&% = CALL ("SMExec_LoadSMPModule",smp$, LEN (smp$),1,h&%(1))
```

At this point we can reset the directory. Prior to loading the program the current directory needs to be that of where the SIMPOL components are located. After the program is loaded the directory can be changed. This is a temporary restriction and it will be removed in later versions.

```


DIRECTORY cd$
```

Now we test the return value from loading the program and if it is not equal to zero then something has gone wrong. If an error occurs here, the likelyhood is that the file was either not found or that a required component could not be loaded. The return values will be SIMPOL error values.

```
  IF (h&% <> 0) THEN
    ? "Error " + STR$ (h&%,".") + " loading '" + smp$ + "'"
  ELSE
```

Only one parameter can be passed to the SIMPOL function and it is always a string, but it is easily possible to place multiple parameters inside the string and to separate them with **TAB** characters. The **TAB**-separated entries will be assigned in order to each argument (which must be of type string) within the function paramter list. The maximum size of the parameter that can be passed when calling from SBL is the size of a Superbase string, which is 4000 characters. When calling from another language like Visual Basic, the limitation would be the maximum size of a VB string. In the case of this example the parameter being passed is the name of a JPEG file from which the size of the image is to be read.

```

    params$ = "SBLOGO.JPG"
```

The maximum size of the return value, which is always a string, is 4000 characters. Here the variable is being presized to accomodate the maximum amount being returned.

```

    res$ = SPACE$ (4000)
```

Now we call the function in the SIMPOL program/library. The return value of the call indicates whether SIMPOL was able to call the function and if the function had any errors. It is not the return value of the SIMPOL function. That is returned in the `res$` parameter. The `h&%` variable receives the return value and the parameters to the function call are the handle to the program/library that we received when it was loaded (`h&%(1)`), the name of the function we are calling in the loaded SIMPOL program/library (`func$`), the length of the function name that we are passing in the previous parameter (`LEN (func$)`), the string parameter that we are passing to the function we are calling (`params$`), the length of the value passed as the parameter (`LEN (params$)`), a variable to hold the return value of our function call (`res$`), the size of the return buffer (`4000`), a pointer or reference to a long integer variable that will receive the number of characters written to the return buffer (`h&%(2)`), and the number of bytes per character that should be used in the return buffer (in the case of SBL this is always `1`).

```

    h&% = CALL ("SMExec_RunSMPFunction",h&%(1),func$,\
                LEN (func$),params$, LEN (params$),\
                res$,4000,h&%(2),1)
    IF (h&% <> 0) THEN
      ? "Error " + STR$ (h&%,"9999");
      ? " executing '" + func$ + "'"
    ELSE
```

Assuming that the function call succeeded, we can retrieve from the return value in `res$` the actual string that may have been returned. For safety's sake we are using the `LEFT$()` function to retrieve the number of characters that we were told was returned in the result. Then we output the results of the function call, once as we received it to show what the return value actually was, and once after having interpreted it. Normally a programmer would decide their own best return format and simply interpret the results as needed.

```
      res$ = LEFT$ (res$,h&%(2))
      ? "Size of the returned result string: ";
      ? STR$ (h&%(2),".")
      ? "Result: '" + res$ + "'"
      pos%% = INSTR (res$," ")
      IF pos%% THEN
        ? "The size of the JPEG image called: " + params$;
        ? " is " + FN numeric( LEFT$ (res$,pos%% - 1));
        ? "x" + FN numeric( MID$ (res$,pos%% + 1))
      END IF
    END IF
```

Now we need to unload the SIMPOL program since we are finished using it. Forgetting to unload the program could result in memory and resources not being freed. Superbase does not free resources on behalf of external programs when it closes. It is the responsibility of the programmer that uses external calls to do that. We pass in the handle to the program (`h&%(1)`) in the unload call. If this handle is incorrect or does not exist, then a GPF can occur, so it is important to maintain these handle values carefully.

```


    h&% = CALL ("SMExec_UnloadSMPModule",h&%(1))
  END IF
```

Finally, we output the finishing time for comparison purposes and clear the registered API calls from memory.

```


  ? :? "End time: " + TIME$ ( NOW ,"hh:mm:ss.sss") + " - ";
  ? DATE$ ( TODAY ,"dd mmm yyyy")

  REGISTER CLEAR

END SUB
```

The preceding program should provide a useful template for building calls to SIMPOL functionality into a program based on SBL. The basic approach should also be clear to anyone working with similar languages such as Visual Basic, although the `REGISTER` command would need to be replaced with the `Declare` syntax.

SMEXEC-Compatible Function In SIMPOL

In the previous section we discussed in-depth a program in SBL that calls a SIMPOL function in a library called `jpeglib.sml`. The actual function in the SIMPOL source code needs to be written to the interface provided by SMEXEC. In this section we will look at that function.

The original function prototype in SIMPOL is as follows:

`function getjpegsize ( string` *`sFilename`*`, integer` *`iWidth`*`, integer` *`iHeight`* `) export`

Unfortunately this function cannot be called via SMEXEC directly, since functions that are called from SMEXEC, as we learned in the previous section, can only take a single string argument and can only return a string result. That means that we will have to create an interface function that can be called via SMEXEC and that can then call the desired function and return the results in the correct format. For ease of understanding later, we can call this function `smexec_getjpegsize()`. The source code for the interface function is shown below:

```
function smexec_getjpegsize(string sFilename) export
  string sResult
  integer iWidth, iHeight, iResult

  iWidth = 0
  iHeight = 0
  iResult = getjpegsize(sFilename, iWidth, iHeight)
  if iResult == 0
    sResult = "w:" + .tostr(iWidth, 10) + " h:" + \
              .tostr(iHeight, 10)
  else
    sResult = "e:" + .tostr(iResult, 10)
  end if
end function sResult
```

In the interface function the only parameter passed in is the *`sFilename`* parameter and the values of the two integers are converted to strings and passed in the return value if the function succeeds and if it fails the error value is passed in the return value. The programmer is free here to implement any method desired to transmit the information via the string result back to the caller.

Part VI. Working with Sockets

One of the more powerful features that is built into SIMPOL is the support for TCP/IP. This includes both client and server components. Using these components it is possible to build a wide range of programs that can interact natively with the Internet. Email clients, web servers, mail servers, file exchange servers and any number of other programs that might require communication over the Internet can be built on top of the SIMPOL sockets support. In this part we will explore basic client and server functionality, which the user can continue to expand upon as required.

Table of Contents

Chapter 16. Client Applications Using TCP/IP

Introduction

The best place to begin when learning about the tcpsocket type is building a simple client program. In this chapter we will examine the code that is used for a basic URL dumping program. It makes a connection to a web server and requests a resource using the GET method of the HTTP protocol. For details of how to program HTTP-compliant applications, the reader is directed to the various RFC's that are associated with this protocol (starting with RFC1945).

Tip

One of the more useful tools when building any type of TCP/IP-based service is the program `telnet`. Telnet provides a console with which the user can examine what is going on when a server connection is made and then can test the interaction with the server. Regardless of whether the objective is to build a server or a client, it is always useful to have a console available with which to test the interaction of the target components.

The tcpsocket Type

SIMPOL's tcpsocket type provides the necessary functionality to create powerful TCP/IP-based programs. The properties of the object are not terribly important for client programs. They are destination and port; both are read-only and contain the destination IP address and port number and the local port number used. More interesting for client programs is the methods: `new()`, `sendblob()`, `sendstring()`, `receiveblob()`, and `receivestring()`. The `new()` method of the tcpsocket type is used to create an object of this type as the result of making a TCP/IP connection to a server using the *`destination`* parameter provided to the `new()` method. The two receive methods are for receiving data and the two send methods for sending data. Either can be used, but the blob versions will normally be more efficient since most protocols over TCP/IP tend to use byte-oriented data and not Unicode.

To Block, or not to Block …

The methods of the tcpsocket type do not block in SIMPOL. However, if no *`timeout`* is specified then the default value of `.inf` will result in the operation never exiting. When waiting on data using either of the receive methods if no data ever arrives the program will wait forever (or until the socket closes). It is far better to control this in the program by setting a timeout value and using it as the optimum time to exit if nothing happens. By placing the receive operation in a loop, the program can continue to loop until no data is received within the allotted timeout period. At that point the program can exit, or in the case of a GUI-style program it can ask the user whether to retry or cancel, etc.

Practical Example — URLDump

The program `urldump.sma` implements a basic HTTP client that permits the sending of a `GET` request to a web server. It then receives the result and outputs the entire returned page into a target file name. Although the same thing can be done with a browser, what makes this program interesting is the fact that it also outputs the headers that were sent from the server, which the browser normally strips off. These are the most interesting part, especially if you are trying to track down why something might be going wrong. The code used in this example could also be reused to eventually provide a light-weight browser component or a web crawling robot or any of a number of different useful programs based on the HTTP protocol.

In the Beginning …

To start with, we need to create a few useful symbolic constants. The use of symbolic constants is what makes a program readable and easy to maintain. Just as using styles in a document makes it easy to change the look and feel of a document very quickly, the same is true of a good computer program.

Example 16.1. Constants portion of the urlget program

```
//////////////////////////////////////
//        Symbolic Constants        //
//////////////////////////////////////

constant sURLID                "://"
constant sSTDPORT              ":80"
constant iTIMEOUT              1000000
constant sGET                  "GET"
constant sSP                   " "
constant sCRLF                 "{d}{a}"
constant sHTTPVER              "HTTP/1.0"
constant sURIBASE              "http://"
constant sCONTENTLENGTH        "content-length:"
constant sHTTP                 "HTTP"
constant sIGNORECHARS          " {d}{a}"

constant sERRTXT_CONNECT       "Failed to connect to host"
constant sERRTXT_SEND          "Error sending request to host, \
                                error number: "
constant sERRTXT_RECEIVINGDATA "Error receiving results from \
                                host, number: "
constant sERRTXT_FILEOPENFAILED "Error opening output file"
constant sERRTXT_PAGE          "Page '"
constant sERRTXT_NOTFOUND      "' not found"
constant sERRTXT_SUCCESS       "' successfully retrieved"
```

There are two types of constants listed in this section, one set consists of the values used in various parts of the program, the other is specifically error and success messages that are returned to the user. Many of these values may not currently make much sense, although the name of the constant may help clarify their meaning and later in the actual program code how they are used will also help clear things up.

The Main Event

Now that we have established the constants we will be using (obviously, these actually got created during the writing and restructuring of the program, not before the work began), let's have a look at the program code.

Example 16.2. Beginning of the `main()` function of the urlget program

```
function main(string sUrl, string sOutfile)
```

```
tcpsocket http
string sDomain, sResult
integer iErrnum, iPos, iContentLength, iPos2
fsfileoutputstream fpo
blob bContent, bReceive, bTmp, bHeader, bStatus

sResult = ""
bTmp = ""

if .instr(sUrl, sURLID) == 0
  sUrl = sURIBASE + sUrl
end if
sDomain = getdomainroot(sUrl)

// Now do a quick check and make sure that if they provided
// a URL like www.foobar.com without the ending slash, that
// we add it.
if sDomain == .rstr(sUrl, .len(sDomain))
  sUrl = sUrl + '/'
end if

if .instr(sDomain, ":") == 0
  sDomain = sDomain + sSTDPORT
end if
```

The start of the program is the `main()` function. It takes two parameters: the URL of the page to retrieve and the name of the output file in which the retrieved page should be stored. After declaring and initializing the variables the program first evaluates the URL and extracts the root domain from it, since tht is what is needed to create the connection. It also checks the URL to ensure that if a base domain was based that the closing slash has been appended (otherwise it adds one) since without the closing slash it will fail when attempting to retrieve the default page from the web server. Finally the root URL is checked to see if it includes the optional port information. If it does (such as `:8080`) then the program does nothing but if there is no port information (the normal case) then it adds `:80` to the end of the domain root. This is necessary since the first parameter to the tcpsocket.`new()` method is the destination in the format of either `IP address:port` or `domain name:port`.

Once the basic initialization has been completed, the program then attempts to open a TCP/IP connection to the web server named in the *`sUrl`* parameter. The variable was named `http` to make clear to anyone reading the source code what the object is used for.

Example 16.3. Creating the socket connection in the urlget program

```

iErrnum = 0
http =@ tcpsocket.new(sDomain, error=iErrnum)
```

If the connection fails, an error message is assigned to the return variable and the program exits. If it is successful, however, then the `GET` request is formulated and sent to the web server via the tcpsocket object referenced via the `http` variable.

Example 16.4. Beginning the TCP/IP conversation in the urlget program

```

if http =@= .nul
  sResult = sERRTXT_CONNECT + sCRLF
else
  // Full-Request and Full-Response use the generic message
  // format of RFC 822 for transferring entities. Both
  // messages may include optional header fields (also
  // known as "headers") and an entity body. The entity
  // body is separated from the headers by a null line
  // (i.e., a line with nothing preceding the CRLF).
  //
  // Full-Request   = Request-Line           ; Section 5.1
  //                  *( General-Header      ; Section 4.3
  //                   | Request-Header      ; Section 5.2
  //                   | Entity-Header )     ; Section 7.1
  //                  CRLF
  //                  [ Entity-Body ]        ; Section 7.2

  //
  // This is known as a full request in the format of HTTP
  // 1.0 but without any additional headers or an entity
  // body, therefore the closing second CRLF to complete
  // the message:
  // Request-Line = Method SP Request-URI SP HTTP-Version
  //                CRLF

  bContent = sGET + sSP + sUrl + sSP + sHTTPVER + sCRLF + \
             sCRLF

  // Although it may not normally be necessary, it is far
  // more elegant to use a socket that will not wait
  // forever. By setting a timeout on the various socket
  // operations (default is .inf -- never) we remain in
  // control of the program, so that if a long time passes
  // with no or insufficient activity, the program can
  // exit properly. In a GUI-style program the user can be
  // asked whether to continue waiting or if they wish to
  // cancel the operation.

  http.sendblob(bContent, timeout=1, error=iErrnum)
```

Assuming that there is no error when sending the request, the program now prepares to receive the response. The program sets up a loop to receive the response from the server. As described earlier, to ensure that the program doesn't hang while waiting for a response (which could happen if the server or the connection went down after the request was sent), the loop is entered and the `receiveblob()` method is called and set to time out when the standard timeout value expires. The loop will only exit if an error occurs, nothing is received on the connection within the scope of the time out period, or the content received contains two carriage-return plus linefeed pairs.

Note

Technically this implementation is not as forgiving as it should be, since according to the standard published in RFC-1945 applications should be reasonably tolerant in terms of which formatting they accept and the carriage return and linefeed pair specifically should be treated as merely linefeed and any carriage return should be dropped (this supports UNIX-based programmers where carriage return is not normally considered to be part of the end of line character).

Example 16.5. Retrieving the header from the web server in the urlget program

```
    if iErrnum != 0
      sResult = sERRTXT_SEND + .tostr(iErrnum, 10) + sCRLF
    else
      bReceive = ""

      // Now we retrieve the header (it may be more than
      // just the header that comes in, but we are technically
      // interested in the header at the moment).
      bHeader = ""
      while
        bTmp = ""
        bTmp = http.receiveblob(timeout=iTIMEOUT, error=iErrnum)
        bHeader = bHeader + .if(bTmp > "", bTmp, "")
        iPos = .inblob(bHeader, .toblob(sCRLF + sCRLF))
      end while iErrnum != 0 or bTmp <= "" or iPos > 0
```

The previous receive loop may or may not have received the entire page but it should have received either the entire header or it exited for some other reason. The next piece of code tests to see if, in fact, it did receive the header and the associated separator. If so, the portion following the header (minus the separator) is assigned to the variable `bReceive` and the header alone is reassigned to the variable `bHeader`.

Example 16.6. Checking the response code in the web page header in the urlget program

```
      // Now that we have received the entire header, we
      // examine the header The first thing to evaluate is
      // the response code, since it needs to be in the 2XX
      // class for success. If it is a 4XX then we won't be
      // getting any content back.

      if iPos > 0
        bReceive = .subblob(bHeader, iPos + 4, .inf)
        bHeader = .subblob(bHeader, 1, iPos - 1)
      end if
```

The next step is to check the header and see what type of response was received from the web server. Unless the web server is using HTTP 0.9 there should be a response code. If there is none, then all we will get back is the body of the response, which will either be the requested page or some error text. If there *is* a full response, then we can evaluate the status line and see if the request succeeded. If it did not, then there is no additional content to retrieve. The `bReceive` variable is set to be equal either to its current value if it has any content or else to the empty string. This is to ensure that the concatenation of the variables later does not result in a value of `.nul`.

Example 16.7. Parsing the web page header in the urlget program

```
      // After receiving and interpreting a request message,
      // a server responds in the form of an HTTP response
      // message.
      //
      //
      //   Response        = Simple-Response | Full-Response
      //
      //   Simple-Response = [ Entity-Body ]
      //
      //   Full-Response   = Status-Line        ; Section 6.1
      //                     *( General-Header  ; Section 4.3
      //                      | Response-Header ; Section 6.2
      //                      | Entity-Header ) ; Section 7.1
      //                     CRLF
      //                     [ Entity-Body ]    ; Section 7.2
      //
      // Status-Line = HTTP-Version SP Status-Code SP
      //               Reason-Phrase CRLF
      // "HTTP/" 1*DIGIT "." 1*DIGIT SP 3DIGIT SP PHRASE CRLF
      //
      // Either we will get a simple response or a full
      // response.


      if .subblob(bHeader, 1, 4) == .toblob(sHTTP)
        iPos = .inblob(bHeader, .toblob(sSP))
        if iPos > 0
          bStatus = .subblob(bHeader, iPos + 1, 3)
        end if
      end if

      if bStatus > "" and bStatus[1] != '2'
        // The page was not found for some reason
        // If the bReceive section is empty, we need to
        // set it to the empty blob (and not .nul) for
        // output later.
        bReceive = .if(bReceive >= "", bReceive, "")
```

Assuming that the request succeeded the next thing to look for is the content length field in the header. Once we either have a content length value or we establish that there is not one to be found, the final step is to read the remainder of the output from the web server. The content length can assist us in deciding when to stop, but it is not necessary, nor is it *always* correct, according to the standard, but for the purpose of this program we will assume that it is.

Example 16.8. Retrieving the web page content in the urlget program

```
      else
        // and look for the "content-length" header field.
        iContentLength = -1
        iPos = .inblob(.toblob(.lcase(bHeader.getstring(1, .inf,\
                      1))), .toblob(sCONTENTLENGTH))
        if iPos > 0
          iPos2 = .inblob(.subblob(bHeader, iPos + \
                                  .len(sCONTENTLENGTH), .inf),\
                         .toblob(sCRLF))
          if iPos2 > 0
            bTmp = .subblob(bHeader, iPos + .len(sCONTENTLENGTH),\
                            iPos2)
            iContentLength = .toval(bTmp.getstring(1, .inf, 1),\
                                    sIGNORECHARS, 10)
          end if
        end if

        // If we found a "content-length" header, then we know
        // how much data is still to come. If we don't, then we
        // can only rely on the timeout and continually loop
        // until we receive nothing on the connection.

        if iContentLength >= 0
          while bReceive.size < iContentLength
            bTmp = ""
            bTmp = http.receiveblob(timeout=iTIMEOUT, \
                                    error=iErrnum)
            bReceive = bReceive + \
                       .if(bTmp > "", bTmp, .toblob(""))
          end while iErrnum != 0 or bTmp <= ""
        else
          while
            bTmp = ""
            bTmp = http.receiveblob(timeout=iTIMEOUT, \
                                    error=iErrnum)
            bReceive = bReceive + \
                       .if(bTmp > "", bTmp, .toblob(""))
          end while iErrnum != 0 or bTmp <= ""
        end if
      end if
```

Now that we have all of the output from the web server (regardless of how much that actually is) it is time to formulate the response to the user, either one of success or failure. Also the output from the web server needs to be written to the output file.

Example 16.9. Returning the results to the user in the urlget program

```
      // Finally, we deal with the result, which is either
      // success or failure. If failure, we need to tell the
      // user what went wrong.

      if iErrnum != 0 and iErrnum != 705
        sResult = sERRTXT_RECEIVINGDATA + \
                  .tostr(iErrnum, 10) + sCRLF
      else
        fpo =@ fsfileoutputstream.new(sOutfile, error=iErrnum)
        if fpo =@= .nul or iErrnum != 0
          sResult = sERRTXT_FILEOPENFAILED + sCRLF
        else
          if bStatus > "" and bStatus[1] != '2'
            sResult = sERRTXT_PAGE + sUrl + \
                      sERRTXT_NOTFOUND + sCRLF
          else
            sResult = sERRTXT_PAGE + sUrl + \
                      sERRTXT_SUCCESS + sCRLF
          end if
          fpo.putblob(bHeader + .toblob(sCRLF + sCRLF) + \
                      bReceive)
        end if
      end if
    end if
  end if
end function sResult
```

Chapter 17. Server Applications Using TCP/IP

Introduction

In the previous chapter we discussed the creation of client applications using TCP/IP. In this one we will explore the other side of the process, a TCP/IP-based server program. In SIMPOL TCP/IP server programs are largely similar to client programs except in one specific area: the initialization of the program. Aside from that, they merely use a tcpsocket object to conduct the opposite half of the conversation to that of the client.

The tcpsocketserver Type

Server programs make use of the tcpsocketserver type for their initialization. The `new()` method takes a *`port`* number and an *`error`* object as parameters. The port number must be in the range from 1 through 65535. More importantly it is advisable that the port should be appropriate for the type of service being supplied. If you are implementing a web server, then port 80 is appropriate. Check for the standard port assignments and try to use one that is either appropriate for a standard service or one that is generally unused. Also, unless you are implementing a service that normally uses a port below 1024, it is strongly recommended that a port in the range from 1025-65535 be selected. Ports between 1 and 1024 are in a range that is typically restricted and they may not as easily pass through a firewall.

Assuming that a tcpsocketserver object is successfully returned from the `new()` method, the only thing left to do is to call the `listen()` method of the object. The `listen` method takes four parameters: a *`function`* reference, a *`reference`* to an object of any type, a *`timeout`* value, and an *`error`* object. The first of these is a reference to the function that is to be called when a connection is made, the second is an optional reference to any object type. The object would typically be some user-defined datatype holding references to resources that are commonly needed by each connection, such as database table references, a reference to the tcpsocketserver object, etc. The *`timeout`* value would probably be set to `.inf` for most systems. The final parameter should by now be familiar. It is an integer object that will be filled with an error value in case anything fails while trying to listen on the port.

When a Connection Occurs

When a client connects to the port on which the server is listening, SIMPOL makes a call to the function referred to by the function reference passed into the `listen()` method. The function must defined with the following prototype:

> *`funcname`* (tcpsocket *`connectionname`*, type(*) *`user-defined-type`*)

The first parameter is the tcpsocket object representing the connection from the caller and the second is the optional user-defined object (this can be `.nul`). Each connection will begin a new thread that starts by calling the function reference passed to the `listen()` method. The conversation with the caller then takes place in exactly the same way that a conversation would take place in a client application (see the previous chapter for details). When the function ends then the thread will also end.

Exiting the `listen()` Method

Calling the `listen()` method of the tcpsocketserver object results in the code in that thread halting at the call to the method until either an error occurs, the timeout expires, or the `break()` method of the tcpsocketserver object is called. At that point, the original thread where the `listen()` method was invoked will continue execution. If that results in the object going out of scope, which would normally destroy the object, that will only happen if no thread is running that was invoked via a connection to the object and no other reference to the object still persists. A connection can be used to shutdown or restart the server if the programmer chooses to implement such functionality. This would require that a reference to the server object also be passed to the function that is called for each connection.

Part VII. User-Interface Components

SIMPOL provides a number of different components for interacting with the user interface. This includes windows, forms, and common dialogs. Some programming environments only provide forms and fail to actually provide windows themselves, but with SIMPOL we have chosen to provide the full range of possible items in order to give the greatest range of functionality to the programmer. User-interface components are the basic building blocks that can make a program easy to use or a nightmare for the user, which of these depends on the choices made by the programmer. When designing a user interface, it is important that the programmer design for simplicity and ease of use. It also helps if they actually have direct contact with those who may be using the interface, to get useful feedback.

Table of Contents

Chapter 18. Using the wxWidgets Component in SIMPOL

In this chapter we will cover the basics of using the wxWidgets support in SIMPOL. Currently this component is available for Win32 and Linux. Other platforms will follow. We have every expectation that the other platforms will come up fairly quickly, since the wxWidgets toolkit is by nature a cross-platform toolkit.

Note

For information about developing data-aware form applications, see Part X, “Programming Data-Aware Form Programs”.

All of the main examples in this chapter are available in the `projects\examples` directory.

Windows and Dialogs

Introduction to Windows and Dialogs

The window support in SIMPOL using the wxWidgets component is designed to allow the creation of a broad variety of windows. Currently we only support the creation of a top-level window or a dialog. The dialog can be either modal or non-modal. The windows can have either sizeable or simple borders; they can have various types of controls such as vertical and horizontal scroll bars (in various styles), the maximize button, the minimize button, the visibility button (this is the close button — it is called the visibility button in SIMPOL because it only affects whether the window is shown or not, it is not destroyed when the user clicks this button, that is left to the programmer). For full details of what is available, see the "wxwindow" entry in the "The wxWidgets-based (WXWN) Components" of the "Components" appendix in the "SIMPOL Language Reference".

Note

If you wish to have scrollbars on the window, or specify which buttons are shown, this must be decided when the window is created, these things cannot be modified after the window has been created!

Windows are not simply provided to house a form, although to start with, that is all that they are able to do. Windows in SIMPOL have been designed with the idea of containership, so that later on, other types of content can be placed into a window without requiring the additional overhead of placing that content on a form. An example might be a terminal or console control that is used to provide a console window with output from the program and input from the user. Another use might be to house a document filter for display of images and other content types in a free-floating window. For this reason it is not a good idea to get into the habit of thinking that windows and forms in SIMPOL are synonymous. It is better to think of a form as one of the types of content that a window can provide.

In order to process the events that occur when a window is shown, there is a special function, called `wxprocess()` that is responsible for processing all of the events that occur in the wxWidgets-based controls in all windows (except for modal dialogs). This function takes a single parameter that tells it how long to process events before returning. In many cases, you may wish to leave the program in this processing state for the entire duration of the program once you have done the initialization and shown the first window. In that case, passing the value `.inf` to the function will keep it processing messages until you call the `wxbreak()` function. As an alternative, you can place the wxprocess() call in a loop and have it exit the call after a set period of time.

Creating a Single Window

It is best to start out working with windows by creating a single standard window. Examine the small example program below (it can be found in the `projects\examples` directory.

Example 18.1. Creating a Single wxwindow

```
function main()
  wxwindow w
  integer iErrnum
  string sResult

  iErrnum = 0
  w =@ wxwindow.new(0, 0, 640, 480, captiontext="Main test window", \
                  error=iErrnum)
  if w =@= .nul
    sResult = "Error number: " + .tostr(iErrnum, 10) + " opening \
              main window{d}{a}"
  else
    w.onvisibilitychange.function =@ quit
    wxprocess(.inf)
  end if
end function sResult


function quit(wxwindow w)
  wxbreak()
end function
```

As can be seen from the above example, the window is created by calling the `new()` method of the wxwindow type. Almost all of the features of the wxwindow object can be specified during creation, and some *must* be specified at that time since they cannot later be changed. The example defaults to showing the window once it has been created, as well as various other features, such as having a system menu, minimize and maximize buttons, making the window sizeable, and so on. The next thing is the assignment of a function to handle the onvisibilitychange event. This event is called when the user takes some action to dismiss the window. Instead of the window being closed, it is hidden. If the programmer has defined a handler for the onvisibilitychange event, then that function will be called and it will be passed the object responsible for the call, in this case the wxwindow object represented by the variable `w` in the `main()` function.

In the example the argument to the `wxprocess()` function was the value `.inf`, so it is essential to assign an event handler to the window's onvisibilitychange event. Otherwise the window will be hidden when the user clicks on the Close button but the thread will continue indefinitely and since there is no longer any method of closing the window the program will not exit unless closed externally (using **Ctrl+C** from the command line or Project → Stop Execution from the IDE).

Creating Multiple Windows

Building upon the beginning made in the previous section, creating additional windows is also quite simple. The following program demonstrates the creation of the main window from the first example followed by the creation of two other windows.

Example 18.2. Example of Creating Multiple Top-Level wxwindows

```
function main()
  wxwindow w, w2, w3, w4
  integer iErrnum
  string sResult

  iErrnum = 0
  w =@ wxwindow.new(0, 0, 320, 240, captiontext="Main test \
                  window", error=iErrnum)
  if w =@= .nul
    sResult = "Error number: " + .tostr(iErrnum, 10) + " opening \
               main window{d}{a}"
  else
    w.onvisibilitychange.function =@ quit
    w2 =@ wxwindow.new(320, 0, 320, 240, captiontext="Second test \
                     window", error=iErrnum,\
                     menubutton=.false, border="simple",\
                     backgroundrgb=0xFFFFFF, vscrollbar=.true,\
                     hscrollbar=.true)
    if w2 =@= .nul
      sResult = "Error number: " + .tostr(iErrnum, 10) + \
                " opening second window{d}{a}"
    else
      w3 =@ wxwindow.new(0, 240, 320, 240, captionbar=.false, \
                       error=iErrnum, vscrollbar=.true)
      if w3 =@= .nul
        sResult = "Error number: " + .tostr(iErrnum, 10) + \
                  " opening third window{d}{a}"
      else
        w4 =@ wxwindow.new(320, 240, 320, 240, \
                         captiontext="Fourth test window", \
                         error=iErrnum, maxbutton=.false, \
                         backgroundrgb=0xC0C0C0)
        w2.onvisibilitychange.function =@ quit
        w3.onvisibilitychange.function =@ quit
        w4.onvisibilitychange.function =@ quit
        wxprocess(.inf)
      end if
    end if
  end if
end function sResult


function quit(wxwindow w)
  wxbreak()
```

```
end function
```

The program above is somewhat more complicated than the earlier program, but the basic technique of creating the windows hasn't changed. Please note that all of the windows are top level windows. The way the program is currently written, closing any of the windows will close all of them. This could easily be modified to use a different (or no function) to handle the closing of all but one of the windows, to prevent the program from exiting except, for example, when the main window is closed. The second window has been created so that it cannot be resized and has no buttons or system menu. It can be closed only using the **Alt**+**F4** keyboard combination, as can the third window, which has no caption and therefore no close gadget or system menu. The fourth window can be resized but not maximized. Most commonly you might actually make the window not resizeable and take away the maximize button if you wanted to prevent any resizing. Two of the windows have the default background window color for the operating system, one has the color white and the last has the color gray.

Working with Dialogs

Dialog windows in SIMPOL are very similar to regular windows. The only significant difference is the fact that dialogs can be shown modally. That means that no further access to the application via the GUI is available as long as the dialog is displayed. The user *must* deal with the modally displayed dialog before they can continue. This is the way most dialogs are displayed. Another type of dialog is the non-modal dialog. These dialogs always stay in front of the windows of the application, but it is possible to click on the windows behind while the dialog is displayed. This sort of dialog is often used for things like a "Find and Replace" dialog in a word processor, where the user should still be able to click into the document while the dialog is displayed. One significant change from the older SBL language is that forms are used both in windows and in dialogs, unlike the older product that differed between forms and dialogs, including having different properties and events for things like editable text controls.

Modal Dialogs

The `wxprocess()` is not used to handle events for a modal dialog, but rather the dialog has a special method for showing itself modally and for handling events while it is shown. The `processmodal()`) method is called to show the dialog and handle events while it is shown. See the examples below:

Example 18.3. A Minimal Modal wxdialog Example

```
function main()
  wxdialog d
  integer e

  e = 0
  d =@ wxdialog.new(1, 1, 600, 400, captiontext="Hello World", visible=.false, error=
  if d !@= .nul
    d.processmodal(.inf)
  end if
end function
```

This dialog program is very simple. Unlike the one for the wxwindow type, there is no `quit()` function and no onvisibilitychange event handler. An important point to note is that the dialog was created with the *`visible`* parameter set to `.false`. It is then shown using the `processmodal()` method of the wxdialog type. Again the value `.inf` is supplied. If the dialog is created with the parameter *`visible`* set to `.true`, then it must be managed by a `wxprocess()` statement.

When a modal dialog is set to invisible, it automatically exits the `processmodal()` method. This happens if the user clicks the close gadget of the window. It can also be done programmatically, in an event handling function called by a button on a form, as shown in the next example.

Example 18.4. A Modal wxdialog

```
function main()
  wxform f
  integer e
  wxdialog d
  wxformbutton b

  e = 0
  f =@ wxform.new(100, 80, 0xC0C0C0, error=e)
  if f !@= .nul
    f.addcontrol(wxformtext, 10, 10, 80, 20, "Hello world!")
    b =@ f.addcontrol(wxformbutton, 30, 55, 40, 22, "OK")
    b.onclick.function =@ quit
    // It is essential to create the dialog invisible, since
    // otherwise it will already be shown and cannot be then
    // shown modally.
    d =@ wxdialog.new(50, 50, innerwidth=f.width, \
                      innerheight=f.height, captiontext="Hello", \
                      visible=.false, error=e)
    if d !@= .nul
      f.setcontainer(d)
      d.processmodal(.inf)
    end if
  end if
end function


function quit(wxformbutton me)
  // Setting a modal dialog to invisible is the only
  // programmatic way to close the dialog.
  me.form.container.setvisible(.false)
end function
```

Note that it is important to create the dialog invisibly. If it is not, then it will cause an error when you try to show it using the `processmodal()`) method, since it is already being shown. To close the modal dialog programmatically, all that is required is to set its visibility back to `.false`.

Non-Modal Dialogs

Non-modal dialogs are created in exactly the same way as modal dialogs. The only significant difference is that non-modal dialogs are simply shown, either by creating them visibly, or by setting them to visible, and their events are handled by the `wxprocess()` function. Here is the same sample modified to be non-modal:

Example 18.5. A Non-Modal wxdialog

```
function main()
  wxform f
  integer e
  wxdialog d
  wxformbutton b

  e = 0
  f =@ wxform.new(100, 80, 0xC0C0C0, error=e)
  if f !@= .nul
    f.addcontrol(wxformtext, 10, 10, 80, 20, "Hello world!")
    b =@ f.addcontrol(wxformbutton, 30, 55, 40, 22, "OK")
    b.onclick.function =@ quit
    // It is still good to create the dialog invisible and then
    // show it after loading the form into it.
    d =@ wxdialog.new(50, 50, innerwidth=f.width, \
                      innerheight=f.height, captiontext="Hello", \
                      visible=.false, error=e)
    // Now we need to trap the onvisibilitychange event (closing
    // the window), since if we don't, the window will close and
    // the program will not.

    d.onvisibilitychange.function =@ quit

    if d !@= .nul
      f.setcontainer(d)
      d.setvisible(.true)
      wxprocess(.inf)
    end if
  end if
end function


function quit(type(*) me)
  // Here we allow dual use of the function
  wxbreak()
end function
```

Dialogs Using Standard Buttons

Another area where dialogs can differ, is that on many operating systems they have a specific look and feel associated with them. Thanks to the cross-platform nature of the wxWidgets library, SIMPOL provides an additional capability to the dialogs, called "Standard Buttons". By using this functionality, the dialog buttons such as OK and Cancel or Yes and No are created and managed by the dialog, rather than being placed by the programmer. This ensures that they are handled correctly for each target platform. For example, on the Macintosh it is customary to put the Cancel on the left, whereas the same button is found on the right on Microsoft Windows.

Note

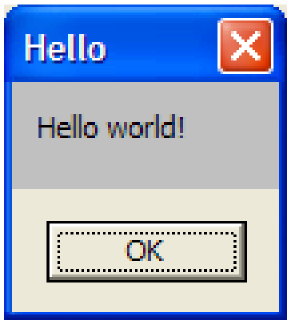
Currently the only style of dialog that will close when the user presses the **Esc** key is the one using standard buttons.

Here is the modal-dialog example converted to use standard buttons:

Example 18.6. A Modal wxdialog with Standard Buttons

```
function main()
  wxform f
  integer e
  wxdialog d

  e = 0
  f =@ wxform.new(100, 40, 0xC0C0C0, error=e)
  if f !@= .nul
    f.addcontrol(wxformtext, 10, 10, 80, 20, "Hello world!")
    // It is essential to create the dialog invisible, since
    // otherwise it will already be shown and cannot be then
    // shown modally.
    d =@ wxdialog.new(50, 50, innerwidth=f.width, innerheight= \
                      f.height, captiontext="Hello", \
                      visible=.false, stdbuttons="ok", error=e)
    if d !@= .nul
      d!ok.onclick.function =@ quit
      f.setcontainer(d)
      d.processmodal(.inf)
    end if
  end if
end function


function quit(wxdialogstdbutton me)
  // Setting a modal dialog to invisible is the only programmatic
  // way to close the dialog.
  me.dialog.setvisible(.false)
end function
```

When you run this program, you may find that it looks a bit strange. That is because the area of the dialog that is controlled by the standard buttons support may be in a different color. The reason for this is that when using the standard buttons, the default font and default system colors are used for the various components, together with where they are supposed to be positioned in the dialog itself.



The wxdialog with standard buttons, showing the color problem.

The solution to this is to use the default system font and default system colors for the form and controls on the form that is being shown in the dialog. To make this easier, there is a supplied library called `uisyshelp.sml` that contains a function called `getdefaultfont()` that returns the default system font as a wxfont object. Another useful item in the library is the syscolors type. Just create an object of this type and call the `new()` method of the type assigning it to the variable, and it will contain an array of all of the available system colors. The total number of colors is found in the count property. A SIMPOL source file called `uisyshelphdr.sma` can be found in the `include` directory. That file contains the symbolic constants for the various colors used on the various versions of Windows. Other operating systems will also eventually be catered for by this library. The colors are stored in the array as sysrgb objects. You can access the individual color components or the entire color value, as shown below:

```
sysrgb color
syscolors colors
integer colorvalue, red, green, blue

colors =@ syscolors.new()
color =@ colors[COLOR_BTNFACE]
colorvalue = color.value
red = color.red
green = color.green
blue = color.blue
```

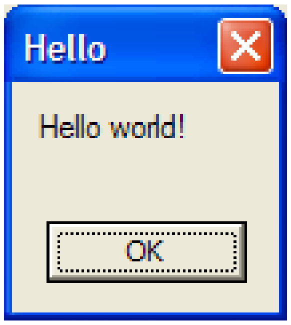
Another useful type is the windowsversion type. It is the return value of the `getwindowsversion()` function. All of the components are available as properties. If all that is needed is a string, the companion function `getwindowsversionstring()` will prove handy. Finally, two more companion functions provide information about the display size. The function `getdisplaysize()` returns the size of the physical display and the `getusabledisplaysize()` returns the size of the display minus the area used by the taskbar.

Using the facilities provided by `uisyshelp.sml` it is possible to rewrite the standard buttons dialog to not have the color problem. Also, to ensure that the font used in the text in the dialog is consistent with that used by the buttons, the default system font is also retrieved. Here is the changed program, please note that it requires the library file to be added to the project settings and the include directory must be added to the include directories section:

Example 18.7. A Modal wxdialog with Standard Buttons Using `uisyshelp.sml`

```
include "uisyshelphdr.sma"
// This requires the uisyshelp.sml library to be added to the project
function main()
  wxform f
  integer e
  wxdialog d
  syscolors colors
  wxfont deffont
  sysrgb color

  e = 0
  colors =@ syscolors.new()
  f =@ wxform.new(100, 40, colors.colors[COLOR_BTNFACE].value, \
                error=e)
  if f !@= .nul
    deffont =@ getdefaultfont()
    f.addcontrol(wxformtext, 10, 10, 80, 20, "Hello world!", font=\
                deffont)
    // It is essential to create the dialog invisible, since
    // otherwise it will already be shown and cannot be then
    // shown modally.
    d =@ wxdialog.new(50, 50, innerwidth=f.width, innerheight=\
                      f.height, captiontext="Hello", \
                      visible=.false, stdbuttons="ok", error=e)
    if d !@= .nul
      d!ok.onclick.function =@ quit
      f.setcontainer(d)
      d.processmodal(.inf)
    end if
  end if
end function


function quit(wxdialogstdbutton me)
  // Setting a modal dialog to invisible is the only programmatic
  // way to close the dialog.
  me.dialog.setvisible(.false)
end function
```

The results can be seen in the image below. The color is now consistent throughout and the font has changed slightly (the letter "H" in "Hello" is taller and the rendering of the "rl" is different).



The wxdialog with standard buttons, without the color problem.

Menu Bars, Menus, and Menu Items

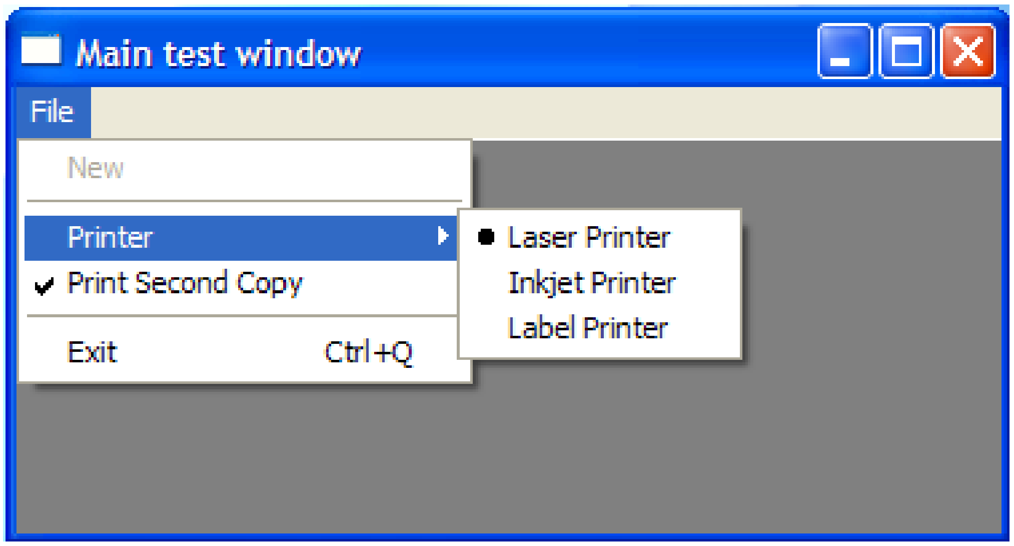
Any reasonably modern user-interface offers various methods of accomplishing the same goals, such as keyboard commands, tool bar and form buttons, and menus. Some user-interface design guides go so far as to say that any functionality that is reachable via a tool bar button or a form button, should *always* provide a menu item to accomplish the same thing. Part of the reason for this is that many users may not be inclined to use a mouse, or under certain circumstances the user may not have a mouse available. Also, providing menus and menu items allows the user to look around in a (hopefully) well-sorted and logically devised set of various groups of functionality. It is an easy way to get to know a product if the menus provide a clear overview of what can be done with the program.

The wxWidgets-based menu support that is part of SIMPOL offers the usual menu capabilities: menus, sub-menus, and menu items that can also be either checkable or one of a group of options. An example menu program is show below:

Example 18.8. A wxmenu Example

```

function main()
  wxwindow w
  wxmenubar mb
  wxmenu filemenu, printmenu
  integer iErrnum
  string sResult

  iErrnum = 0
  w =@ wxwindow.new(0, 0, 640, 480, captiontext="Main test \
                  window", error=iErrnum)
  if w =@= .nul
    sResult = "Error number: " + .tostr(iErrnum, 10) + \
              " opening main window{d}{a}"
  else
    w.onvisibilitychange.function =@ quit

    mb =@ wxmenubar.new()
    filemenu =@ wxmenu.new()
    mb.insert(filemenu, "&File", name="filemenu")
    filemenu.insert("", "&New", enabled=.false, name="new")
    filemenu.insert("separator")

    printmenu =@ wxmenu.new()
    printmenu.insert("radio", "&Laser Printer", checked=.true, \
                     name="laserprinter")
    printmenu.insert("radio", "&Inkjet Printer", \
                     name="inkjetprinter")
    printmenu.insert("radio", "La&bel Printer", \
                     name="labelprinter")

    filemenu.insert("submenu", "&Printer", submenu=printmenu, \
                    name="printmenu")
    filemenu.insert("checkable", "Print Second &Copy", \
                    checked=.true, name="secondcopy")
    filemenu.insert("separator")
    filemenu.insert("", "E&xit{9}Ctrl+Q", name="exit")
    filemenu!exit.onselect.function =@ quit

    mb.setwindow(w)
    wxprocess(.inf)
  end if
end function sResult


function quit(type(*) me)
  wxbreak()
end function
```

The program above demonstrates most of the capabilities of the menu support in SIMPOL. A menubar (like all other SIMPOL GUI objects) exists indepently of its representation in a window. A menubar can be assigned to a window and then be replaced by another. Looking at the preceding program, we first create a menu bar and then an empty menu that we insert into the menu bar. It is not necessary to do it this way, we could just as easily have filled the menu first and then inserted it into the menu bar. Next we insert a menu item that is set to disabled from the start. Following this, a separator is inserted and then a printer menu is created that contains three radio items, only one of which can be selected and we pre-select the first one. The printer menu is then inserted into the file menu as a sub menu. This is then followed by a checkable item, entitled "Print Second Copy". Next we add another separator and the item to exit the program. This item is also given a keyboard accelerator (by adding a tab character and the desired accelerator combination). In closing the `quit()` function is assigned to the onselect event of the exit item. Finally the menu bar is set into the window and the program then waits for events. A picture of the menu can be seen below:



An example of the wxmenu in action.

The trickiest part of working with menus may be learning how to correctly address the various parts. The containership model of the menus is as follows: wxmenubar contains objects of type wxmenubarentry. That contains objects of type wxmenu. The wxmenu objects contain objects of type wxmenuitem, which can themselves contain objects of type wxmenu. So, assuming that there is a variable called `mb` and the menu bar from the program above, to access the `labelprinter` item, the following code would be used:

```
mb!filemenu.menu!printmenu.submenu!labelprinter
```

The member operator (!) is used to access the wxmenubarentry that contains a reference to the `filemenu` menu object. The member operator is again used to access the wxmenuitem represented by the `print-menu` object, and then accesses the `labelprinter` item of the sub menu by using the member operator on the submenu property of the `printmenu` menu item.

Creating menus by hand can be quite boring, but until there is a menu editor available, the menu editor provided by the older Superbase product (including the downloadable demo) can be used, together with the conversion utility provided for converting Superbase menu programs to SIMPOL source code. The menu conversion utility is in the `utilities` directory and the program is called: `ngmengen.sbp`.

Forms and Form Controls

Now that we have windows, dialogs, and menus, it might be useful if we had something to actually put into the windows. That is where forms and form controls come in. This is the primary user-interface area that makes up the real "meat" of most applications.

Introduction to Forms

The form capability in SIMPOL is intended to provide the primary interface support for creating desktop applications. Forms are normally contained in windows or dialogs, so make sure that you read the section called “Windows and Dialogs” before trying to make use of the content of this section. Currently the following form control types are available for use on a form:

- wxformtext — for providing labels and other text on the form
- wxformedittext — provide controls for data-entry from the user
- wxformbutton — pushbuttons for every occasion
- wxformbitmapbutton — pushbuttons with images instead of text
- wxformcheckbox — for getting individual choices from the user
- wxformoption — to get one of a set of choices from the user
- wxformcombo — to allow the user to select a single choice from a drop-down list or to enter a new option
- wxformlist — to get one or more selections from a list of options

All of the form controls have the type tag `wxformcontrol` so that a variable can be created that can hold a reference to any valid form control.

Creating Simple Forms

The form has a background color, and a height and width. It also has a container property that holds a reference to the current container object (or `.nul`). Forms and form controls exist outside of their visible representation. This is quite handy, since a form does not need to be displayed or even be associated with a container and in spite of that, the contents of the various controls can be assigned or read. A form can be moved from one window to the next simply by calling the `setcontainer()` method of the form and passing the target window reference to the method.

To assign a color to a form, window, or form control, SIMPOL makes use of the rgb type. This provides a method of assigning a color using the standard RGB methodology. An rgb object has as its value the combined integer value of the color. This can also be used to assign colors to controls and forms, etc. but in general it is better to use an rgb object since under certain conditions the requested color may not be available and when the color is created using the rgb type the closest available color will be selected.

Creating a basic form is quite simple, as can be seen from the following program code:

```
function main()
  string sResult
  wxform f
  integer iErrnum
  wxwindow w

  // Initialize the error variable so that we are passing a valid
  // object to be filled.
  iErrnum = 0

  // Create our initial form
  f =@ wxform.new(640, 400, error=iErrnum)
  if f =@= .nul
    sResult = "Error " + .tostr(iErrnum, 10) + \
              " creating form{d}{a}"
  else
    // Assuming the form was created successfully, assign a color
    // to the background. Here, since we are creating the rgb
    // object from pure values, we should use the red, green, and
    // blue properties to ensure that a valid color is produced.
    f.setbackgroundrgb(red=0xa0, green=0xa0, blue=0xa0)

    // Create a window to contain the form
    w =@ wxwindow.new(0, 0, 640, 480, captiontext="Test form \
                      window", error=iErrnum)
    if w =@= .nul
      sResult = "Error " + .tostr(iErrnum, 10) + " creating \
                window{d}{a}"
    else
      // Assuming the window was created successfully, assign the
      // function to handle someone clicking the close gadget.
      w.onvisibilitychange.function =@ quit
      // Now we assign the background color from the form to the
      // window so that they match. This time we can assign the
      // rgb object since it will be certain to be a valid color.
      w.setbackgroundrgb(f.backgroundrgb)

      // Now let's add a minimal title to the form, roughly
      // centered
      f.addcontrol(wxformtext, 285, 19, 70, 16, "Test Form")

      // Finally, we move the form into the window that we have
      // prepared
      f.setcontainer(w)

      // Enter the wxprocess() function and wait for events
      // The user pressing the close gadget (or Alt+F4) will cause
      // the onvisibilitychange method to fire which will then
      // call the wxbreak() function. That will cause it to drop
      // out of the wxprocess() function, ending the program.
      wxprocess(.inf)
      sResult = "Success{d}{a}"
    end if
  end if
end function sResult


function quit(wxwindow w)
  wxbreak()
end function
```

The majority of what is going on can be read from the program comments in the code. Note that the form creation actually takes place before the window creation. This is a very basic form program that doesn't have all that much happening. Still, it is an excellent little program to study, since it teaches a number of very important concepts when creating forms. It is also a good idea to read the section in the "SIMPOL Language Reference" regarding the form and each of the form controls. The form's `addcontrol()` method has been designed to allow virtually all of the various properties for each control to be set when the control is created. This will allow the program code to be compacter for those who prefer such an approach. Many people complained about the *wordiness* of the object SBL code.

Although it is a little early in the life-cycle of the SIMPOL language to talk about *best practice* there are certain things that are standard with most event-driven programs that can already be applied here. Most programs can be broken down into three pieces: initialization, execution, and termination. Initialization sets up the program, execution runs the program, and termination closes the program performing any necessary cleanup. From this we can see that the previous program is initialized, then enters the `wxprocess()` function and remains there processing events until some event results in the `wxbreak()` function being called. The termination of the program is quite minimal since SIMPOL tends to cleanup most things for you, all that is left is assigning the return value of the program and returning it.

Working with Form Controls

Now that we have had a chance to work a bit with forms, the next step is to add more controls to the form. As such, the next program `wxforms2.smp` builds on the work done in the first program. It will look a bit more complicated, but not too much more.

```
function main()
  string sResult
  wxform f
  integer iErrnum
  wxwindow w
  type(wxformcontrol) fc
  wxfont font

  // Initialize the error variable so that we are passing a valid
  // object to be filled.
  iErrnum = 0

  // Create our initial form
  f =@ wxform.new(640, 400, error=iErrnum)
  if f =@= .nul
    sResult = "Error " + .tostr(iErrnum, 10) + " creating form{d}{a}"
  else
    // Assuming the form was created successfully, assign a color to
    // the background. Here, since we are creating the rgb object
    // from pure values, we should use the red, green, and blue
    // properties to ensure that a valid color is produced.
    f.setbackgroundrgb(red=0xc0, green=0xc0, blue=0xc0)

    // Create a window to contain the form
    w =@ wxwindow.new(0, 0, 640, 480, captiontext="Test form \
                  window", error=iErrnum)
    if w =@= .nul
      sResult = "Error " + .tostr(iErrnum, 10) + " creating \
                window{d}{a}"
    else
      sResult = "Success{d}{a}"

      // Assuming the window was created successfully, assign the
      // function to handle someone clicking the close gadget.
      w.onvisibilitychange.function =@ quit
      // Now we assign the background color from the form to the
      // window so that they match. This time we can assign the rgb
      // object since it will be certain to be a valid color.
      w.setbackgroundrgb(f.backgroundrgb)

      // Now let's add a minimal title to the form, roughly centered
      fc =@ f.addcontrol(wxformtext, 285, 19, 70, 16, "Test Form")
      fc.setbackgroundrgb(red=0xC0, green=0xC0, blue=0xC0)
      fc.settextrgb(red=0x0, green=0x0, blue=0x0)
      font =@ wxfont.new("Arial", 10)
      fc.setfont(font)

      // Now a few more controls just for playing with
      // a place to type:
      fc =@ f.addcontrol(wxformtext, 39, 84, 98, 16, "Editable text\
                         box")
      fc.setbackgroundrgb(red=0xC0, green=0xC0, blue=0xC0)
      fc.settextrgb(red=0x0, green=0x0, blue=0x0)
      fc.setfont(font)
      fc =@ f.addcontrol(wxformedittext, 146, 82, 175, 20)
      fc.setbackgroundrgb(red=0xFF, green=0xFF, blue=0xFF)
      fc.settextrgb(red=0x0, green=0x0, blue=0x0)
      fc.setfont(font)

      // some option buttons
      fc =@ f.addcontrol(wxformoption, 146, 132, 63, 22, "one")
      fc.setbackgroundrgb(red=0xC0, green=0xC0, blue=0xC0)
      fc.settextrgb(red=0x0, green=0x0, blue=0x0)
      fc.setfont(font)
      fc =@ f.addcontrol(wxformoption, 146, 154, 63, 22, "two")
      fc.setbackgroundrgb(red=0xC0, green=0xC0, blue=0xC0)
      fc.settextrgb(red=0x0, green=0x0, blue=0x0)
      fc.setfont(font)
      fc =@ f.addcontrol(wxformoption, 146, 176, 63, 22, "three")
      fc.setbackgroundrgb(red=0xC0, green=0xC0, blue=0xC0)
      fc.settextrgb(red=0x0, green=0x0, blue=0x0)
      fc.setfont(font)

      // a check box
      fc =@ f.addcontrol(wxformcheckbox, 148, 107, 70, 22, "Do it")
      fc.setbackgroundrgb(red=0xC0, green=0xC0, blue=0xC0)
      fc.settextrgb(red=0x0, green=0x0, blue=0x0)
      fc.setfont(font)

      // a combo box
      fc =@ f.addcontrol(wxformcombo, 148, 210, 172, 126, edittype=\
                         "droplist")
      fc.setbackgroundrgb(red=0xFF, green=0xFF, blue=0xFF)
      fc.settextrgb(red=0x0, green=0x0, blue=0x0)
      fc.setfont(font)
      fc.insert("red")
      fc.insert("green")
      fc.insert("blue")
      fc.insert("yellow")
      fc.insert("magenta")
      fc.insert("cyan")
      // a list box
      fc =@ f.addcontrol(wxformlist, 381, 103, 182, 217, \
                         selectiontype="extended")
      fc.insert("red")
      fc.insert("green")
      fc.insert("blue")
      fc.insert("yellow")
      fc.insert("magenta")
      fc.insert("cyan")
      fc.setbackgroundrgb(red=0xFF, green=0xFF, blue=0xFF)
      fc.settextrgb(red=0x0, green=0x0, blue=0x0)
      fc.setfont(font)
      fc =@ f.addcontrol(wxformtext, 381, 85, 117, 16, "Choose one\
                         or more")
      fc.setbackgroundrgb(red=0xC0, green=0xC0, blue=0xC0)
      fc.settextrgb(red=0x0, green=0x0, blue=0x0)
      fc.setfont(font)
      fc =@ f.addcontrol(wxformtext, 69, 212, 69, 16, "Choose one")
      fc.setbackgroundrgb(red=0xC0, green=0xC0, blue=0xC0)
      fc.settextrgb(red=0x0, green=0x0, blue=0x0)
      fc.setfont(font)

      // and some buttons: an OK and a Cancel button
      fc =@ f.addcontrol(wxformbutton, 226, 352, 83, 27, "OK")
      fc.setfont(font)
      // here we assign the event handling function
      fc.onclick.function =@ evaluateform
      // and here we assign a reference to the object we want passed
      // to the function.
      fc.onclick.reference =@ sResult
      fc =@ f.addcontrol(wxformbutton, 334, 352, 83, 27, "Cancel")
      fc.setfont(font)
      fc.onclick.function =@ evaluateform
      fc.onclick.reference =@ sResult

      // Finally, we move the form into the window that we have
      // prepared.
      f.setcontainer(w)

      // Enter the process() function and wait for events
      // The user pressing the close gadget (or Alt+F4) will cause
      // onvisibilitychange method to fire which will then call the
      // the wxbreak() function. That will cause it to drop out of
      // the wxprocess() function, ending the program.

      // Also, clicking on either of the wxformbutton controls will
      // call the evaluateform() function which in turn will read the
      // contents controls and assign that plus which button was
      // pressed to the string of the object referred to by sResult.
      // Then it will call the wxbreak() function with the same result
      // as above.
      wxprocess(.inf)
    end if
  end if
```

```
end function sResult


function quit(wxwindow w)
  wxbreak()
end function


function evaluateform(wxformbutton b, string s)
  type(wxformcontrol) fc
  integer i
  boolean bDoneone

  if b.text == "Cancel"
    s = "You pressed Cancel{d}{a}"
  else
    s = "You pressed OK{d}{a}"
  end if

  fc =@ b.form.firstcontrol
  while
    if fc.type =@= wxformedittext
      s = s + "You typed: " + fc.text + "{d}{a}"
    else if fc.type =@= wxformcheckbox
      s = s + "Do it was " + .if(fc.state == "", "not ", "") + \
          "checked{d}{a}"
    else if fc.type =@= wxformoption
      if fc.state == "on"
        s = s + "You selected the option '" + fc.text + "'{d}{a}"
      end if
    else if fc.type =@= wxformlist
      i = fc.getselected(1)
      bDoneone = .false
      if i == .nul
        s = s + "You did not select an entry from the list{d}{a}"
      else
        while
          if not bDoneone
            s = s + "You selected the following items from the \
                list:{d}{a}"
            bDoneone = .true
          end if
          s = s + "  '" + fc[i] + "'{d}{a}"
          i = fc.getselected(i + 1)
        end while  i == .nul
      end if
    else if fc.type =@= wxformcombo
      if fc.text >= ""
        s = s + "You selected the entry '" + (fc.text) + "' from \
            the combo box{d}{a}"
      else
        s = s + "You did not select an entry from the combo \
            box{d}{a}"
      end if
    end if
    fc =@ fc.next
  end while fc =@= b.form.firstcontrol
  wxbreak()
end function
```

As we can see from the previous program listing, although it is somewhat longer than the first example, it is not greatly different. It simply contains more controls and an additional event handling function. The technique for exiting the `wxprocess()` function is identical to the one used by the `quit()`. In both cases the `wxbreak()` function is called.

Creating the form control information by hand could be quite long-winded, but fortunately the content shown in the program was not created by hand, but rather generated using a utility program from a Superbase form created with the Superbase Form Designer. Until a form designing utility has been created for SIMPOL-based forms, we will use the older Superbase Form Designer to provide this functionality. The current utility can convert all of the supported form objects including the event procedure names. The actual utility is more sophisticated than the example code shown here. It also creates an array of controls that is indexed according to the original control name from the Superbase form and which provides the form control reference as the array element. This makes it fairly easy to work with the form controls since they are all placed into the form controls array. Later in the life-cycle of the forms, this will probably not be needed but the technique will still work and be valid. The form conversion utility is in the `utilities` directory and the program is called: `sbv2wxsm.sbp`.

The Grid Control

In this section we will look at the grid control. In SIMPOL one of the new controls that has been added to the mix is a general purpose grid control. The grid control can be used for any number of things. Internally in SIMPOL we will use it for implementing record and table view, for creating and modifying database table definitions, as the properties grid for the form and report designers, and much more. The grid functionality is currently a moving target, so this section will be regularly revisited during the pre-release cycle as new capabilities are added to the grid control. The most current information will always be found in the SIMPOL Language Reference book.

Let's build a little sample program, like the other programs that went before, to play a bit with the grid control. As is the case with the other sample programs, this program will always be found in the `projects\examples` directory. This program will be called `wxgrid.sma`.

```

function main()
  wxform f
  wxwindow w
  wxformgrid g
  integer e
  string s

  s = ""
  e = 0
  w =@ wxwindow.new(0, 0, 800, 600, visible=.false, \
                    captiontext="wxgrid example", \
                    border="simple", maxbutton=.false, error=e)
  if w =@= .nul
    s = "Error " + .tostr(e, 10) + " creating window{d}{a}"
  else
    // Assign the function to handle the user clicking the close gadget
    w.onvisibilitychange.function =@ quit
    // Create a new form
    f =@ wxform.new(w.innerwidth, w.innerheight, 0xC0C0C0, error=e)

    if f =@= .nul
      s = "Error " + .tostr(e, 10) + " creating form{d}{a}"
    else
      // Add the grid control to the form
      g =@ f.addcontrol(wxformgrid, 1, 1, f.width - 2, f.height - 2,\
                        rowcount=50, colcount=30, error=e)
      if g =@= .nul
        s = "Error " + .tostr(e, 10) + " creating grid{d}{a}"
      else
        // Change some of the labels, just to show we can
        g.setcollabels(startcol=1, "a column label", "b", "3", "d", \
                       "5", "Foo", "gosh!")
        g.setrowlabels(startrow=1, "a very long row label", "bb", \
                       "C", "IV")
        // Increase the width of the row labels to accomodate the
        // long one
        g.setrowlabelwidth(130)
        // Now we create a cell with a set of choices (combo box),
        // we could easily assign this to multiple cells in the
        // same statement
        g.setcellchoices(row=2, col=2, allowothers=.false, \
                         "<pick one>", "United Kingdom", \
                         "United States", "Germany", "France", \
                         "Italy", "Sweden", "Spain", "Portugal", \
                         "Norway", "Denmark", "Belgium", \
                         "Netherlands", "Luxembourg", "Greece", \
                         "Ireland", "Austria", value="<pick one>")
        // Assign some normal text content to one cell
        g.setcellvalue(3, 3, "This is read only")
        // Now make that cell read only
        g.setcellreadonly(3, 3, .true)
        // Widen the first column just to show we can
        g.setcolwidths(col=1, 190)
        // Assign the same text to a bunch of cells in a range
        g.setcellvalue(startrow=5, endrow=10, startcol=4, endcol=5, \
                       value="This is sooooo cool!!")
        // Widen the two columns to which we assigned the text
        g.setcolwidths(startcol=4, endcol=5, colwidth=130)
        // Assign a multi-line text. Currently it is not possible
        // to edit multiline text and allow the entry of new line
        // characters.
        g.setcellvalue(row=7, col=1, \
                       value="This is a text that goes over{a}\
                       multiple lines. We will also increase{a}\
                       the height of the row to compensate")
        // Now we can increase the height of the row to show the
        // multiline text
        g.setrowheights(row=7, rowheight=60)
        // Place the form into the window
        f.setcontainer(w)
        // Now show the window and wait for events
        w.setstate(visible=.true)
        wxprocess(.inf)
      end if
    end if
  end if
end function s


function quit(wxwindow w)
  wxbreak()
end function
```

There are numerous other things that we can do with the grid control, this is only a very simple example. More capabilities will be added gradually as they become necessary, but already using the current state of the grid control, a great deal can be done, just use your imagination!

Summary

In this section we have looked at the basics of working with forms in SIMPOL. For more information about the specifics of working with each individual control, see the appropriate sections in the "SIMPOL Language Reference". Working with the forms is not greatly different than working with forms in the older Object SBL language. The event type that is embedded in the various controls is similar to the events from the older language. One notable difference is that an optional reference property, which must be an object reference, can be assigned. This provides a solution to the question of how to pass important information into the event handling function. If more than one piece of information is required then a user-defined type can be created that consolidates all of the information that is needed into a single object that is then passed around as required. A major advantage of this approach is that if later more information is needed, the interface of the functions need not be changed, merely the definition of the type needs to be expanded to include the additional information.

Common Dialogs

An important part of working with a graphical user interface is the area known as Common Dialogs. These are the dialogs that are presented to the user to provide access to features that are common across applications, thus retaining a standard look and feel. These types of dialogs include: file selection (for opening or saving files), directory selection (with or without a new directory button), message boxes, color selection, font selection, page setup, printer setup, and progress meters (gauges). All of these are available through wxWidgets and will eventually be part of the capabilities provided by SIMPOL. Most of these will be provided as functions, such as the `wxfiledialog()`, the `wxdirectorydialog()`, and the `wxmessagedialog()` functions. The use of each is quite straightforward, so check out the associated information in the "SIMPOL Language Reference" book. These common dialogs replace the ones provided by the older UTUI component, which is deprecated. For further information regarding the Common Dialog support, see Chapter 19, *Common Dialogs and Other UI Utilities in SIMPOL*.

Parting Notes

In this chapter an attempt has been made to introduce you to the various graphical user interface components that are currently available in SIMPOL. The example do not include the message box and common dialog components (yet), though those can be found in a preliminary form in Chapter 19, *Common Dialogs and Other UI Utilities in SIMPOL*. For a fairly thorough and more in-depth program code example, examine the `demo` project in the `projects\forms` directory.

Chapter 19. Common Dialogs and Other UI Utilities in SIMPOL

Common dialogs are typically provided by the operating system to perform commonly required tasks. Typical common dialogs would be those for picking a file to open or to save, for configuring the printer, or selecting a font or color. Currently SIMPOL provides support for the selecting of a file to open and to save, presenting a message to the user, and the selection of a directory (with or without a new directory button). Others will follow as we progress. The look and feel of these dialogs including largely the functionality that is provided is very operating system dependent, so it may be that certain capabilities are not provided that might otherwise be possible, in order to retain a common user-interface cross-platform. The message box is a good example. This may have a different name on different platforms but the functionality is fairly consistent: a dialog window is presented with a message to the user, possibly including an icon, and one or more buttons from which the user must choose.

Note

Please come back and check this chapter regularly in each release. The common dialog support is a currently moving target. The older UTUI components that provided the initial common dialog support are now deprecated! Please migrate to the newer wxWidgets-based common dialog support.

Common Dialogs in SIMPOL

Common dialogs provide a method of accessing standard user-interface components that are provided to allow a common look and feel. These are implemented using functions rather than types in most cases. The following functions are currently implemented:

- `wxfiledialog()` — for getting a file name for opening or saving
- `wxdirectorydialog()` — for getting the name of a directory, including allowing the user to create a new one
- `wxmessagedialog()` — for showing some information to the user and getting their response (one of `"ok"`, `"cancel"`, `"yes"`, or `"no"`)

The `wxfiledialog()` function implements the functionality that allows the user to select an existing file to be opened as well as that of selecting the path and then typing in a file name for a file to be saved. The behavior of the dialog depends on the purpose for which it is being used. If it is being used to select a file name for saving and the user selects a file that already exists, and if the style includes the value `"overwriteprompt"` then they will automatically be prompted for confirmation that they wish to overwrite the file that they have selected. In some operating systems, if the user enters the name of a file that does not exist when using this technology to open a file, they will be prompted with the question of whether they wish to create a new file. In other cases they may not be able to open a file that does not exist. The style value `"mustexist"` plays a role here.

The full syntax of this function is:

`wxfiledialog` (type(wxdialogparent) *`parent`*, string *`message`*, string *`defaultdirectory`*, string *`defaultfilename`*, string *`wildcard`*, string *`style`*, string *`filename`*, string *`result`*)

Currently the *`style`* can contain either the value `"open"` or `"save"` to decide the basic type of file dialog. In addition to these two values, the style values `"mustexist"`, `"overwriteprompt"`, and `"multiple"` are provided to further influence how the dialog works. Generally, the viable combinations are: `"open,mustexist"`, `"open,multiple"`, `"open,mustexist,multiple"`, and `"save,overwriteprompt"`. The *`wildcard`* parameter provides the capability to have a number of different extensions and file descriptions on some platforms. However, not all platforms support this capability so an application should not rely on this ability in a cross-platform environment. A default value for the file name can be provided using the *`defaultfilename`* parameter. The starting directory is defined normally by the *`defaultdirectory`* parameter, but this is a fairly complicated issue, so check the description of this parameter in the "SIMPOL Language Reference" book. The *`filename`* and *`result`* parameters must be actual objects, since they will be filled with a value by the function. The *`result`* parameter will contain either `"ok"` or `"cancel"` indicating the action taken by the user. Although a number could have been chosen, the decision was taken that using strings for the return value is more programmer-friendly and that in the majority of places that they will be used, they would not be difficult strings for most programmers to understand. They can still be assigned to constants if desired by the programmer. For complete documentation on the features of this function, see the "wxWidgets" section of the "Components" chapter in the "SIMPOL Language Reference" book.

Note

SBL programmers should note that this function corresponds to the **REQUEST** command in SBL for types 26 and 27.

The syntax and usage of the `wxdirectorydialog()` function is very similar to that of the preceding one. It is less complex, since the number of options is less. It is used to retrieve the name of a directory and can optionally provide the user with the ability to create a new directory.

Note

SBL programmers should note that this function corresponds to the **REQUEST** command in SBL for type 28.

Message Boxes in SIMPOL

An extremely common requirement in programming is to be able to communicate with the user via a dialog box that displays a message. The dialog box should normally not permit the user to continue until they have responded to the message. Most operating systems provide this type of functionality although they differ in the details of how many different styles of message box may exist and therefore how many different icon types or button combinations can be provided.

The message box function can be very simple. As little as:

```
string sResponse
sResponse = .nul
wxmessagedialog(.nul, "Hello world!", "Message from SIMPOL", \
                "ok", result=sResponse)
```

It can also be as complicated as:

```
string sResponse
sResponse = .nul
wxmessagedialog(.nul, "Hello world!", "Message from SIMPOL", \
            "yesno_defaultno", "question", result=sResponse)
```

This example shows a Yes and a No button and also shows an icon indicating the purpose of the message. Currently there are only six possible values for the icon, `""`, `"question"`, `"error"`, `"exclaim"`, `"hand"`, or `"information"`. For full information on the various parameters see the section covering the `wxmessagedialog()` function in the "SIMPOL Language Reference".

Part VIII. Converting From SBL

This part is dedicated to discussing the similarities and differences between SBL and SIMPOL in an effort to ensure that moving applications and programming knowledge from the older Superbase product line to the newer is as painless as possible. This part will also include information about program code and utility programs designed to assist in the conversion process.

Table of Contents

Chapter 20. Moving from SBL to SIMPOL

Making the move from being a traditional Superbase SBL programmer to being a SIMPOL programmer doesn't need to be as complicated as many people might believe. Although SIMPOL is an object-oriented programming language, it is not nearly as complex or difficult as learning Java, C#, or even VB.NET. There are very few key words and no real commands in SIMPOL. In SBL there are literally hundreds of key words and a very complicated set of parameters that can be passed to each command. If your SBL programs begin with a **SUB main()** and tend to be event-driven, spending most of the time in a loop waiting for the user to do something, then writing programs in SIMPOL won't be terribly complicated for you, but even if you have gotten into the habit of just using global variables and **GOTO**, **GOSUB**, and **RETURN**, it is still possible to learn to write programs in SIMPOL without too much effort. In the latter case, the job is complicated somewhat by needing to learn to use structured programming techniques and do some advance planning before writing the program, but the benefits are considerable: easier to understand code, easier and faster maintenance, and a greater amount of code reuse resulting in smaller programs and an ever-growing toolbox of useful functions (and types).

The Basics

It is probably useful to discuss the available data types and programming elements, and then have a look at the various commands from SBL and see how they are done using SIMPOL. In SBL there are four basic data types: strings (maximum length 4000 characters), short integers (-32,768 to +32,767), long integers (-2,147,483,648 to +2,147,483,647), and IEEE double-precision floating point numbers — decimal values — (± $10^{-323.3}$ to $10^{308.3}$). Variables of these types are indicated by using the dollar sign ($) for strings, a percent symbol (%) for auto variables (can hold any of the numeric types), two percent symbols (%%) for short integers, the ampersand and percent characters (&%) for long integers, and either the hash and percent (#%) or exclamation mark and percent (!%) characters for decimal values. In SIMPOL there is also a string type (maximum length is limited by memory), an integer type (greater degree of significant digits than an SBL HugeInteger), a number type (exact precision, not floating point and virtually unlimited size), a boolean type (true and false values), and the blob type (virtually unlimited in size array of bytes). SIMPOL does not use any characters to indicate data type for variables so it is generally a good idea to use some sort of convention, such as a leading `s` character for strings, `i` for integers, `b` for booleans, `bl` for blobs, and `n` for numbers. Which convention is used is not as important as simply picking one and being consistent in its use. It is even okay to use `b` for both blobs and booleans if it is obvious which is which.

One of the biggest and most significant differences between SBL and SIMPOL is the capability in SIMPOL to create user-defined data types. This capability can completely change the approach to solving a problem. It also makes it possible to use a much more object-oriented approach to solving a problem, though it is not required to do so. User-defined types can include properties that are of any of the standard types, be references to other objects, or be actual embedded objects of some other complex type including user-defined types. This allows for fairly complex object design, which can help to solve many problems that in SBL programs would only be able to be solved by using sets of variables and variable arrays. One of the biggest advantages to using user-defined types is that when passing information from one function to another, the interface does not need to change if one more piece of information is required. Instead the type is changed and the information is added to the object that is being passed, so no change to the parameter list of the function is necessary.

SIMPOL has a very small set of key words, which are listed below:

- and
- AND

- else
- embed
- end
- export
- function
- if
- mod
- not
- or
- OR
- reference
- resolve
- type
- while
- XOR

It also has a useful set of operators, which are summarized in the following list:

- :, ;
- ,
- +
- -
- *
- /
- ==
- >
- <
- >=
- <=
- <>, !=
- @=, =@
- =@=
- !@=, <@>
- ()

- []
- {}
- ",'
- //
- "",''
- \

For a complete description of the key words and operators please see the SIMPOL Language Reference. Unlike classic SBL, SIMPOL has an extremely small set of key words, a slightly larger set of operators, a number of intrinsic and system functions, and an ever growing number of components, free functions and types (some of which are created using SIMPOL itself). In SBL there is a large number of key words, some of which are operators, some of which are commands, and some of which are functions. There is also a fairly large set of objects that model a number of the components of the system, such as the forms, form controls, and windows, but which do not have a representation for the database files, fields, indexes, and records. In SIMPOL everything that is not a key word, an operator, an intrinsic function, or a system function, is a type and to work with the type it provides built-in methods (functions) and in some cases allows the assignment of event handling functions. In the next section we will compare the key words that actually represent the underlying language structure and in a later section we will explore the special commands and functions.

Comparison Between Language Primitives in SIMPOL and SBL

The following table contains a comparison between the language primitives in SBL and SIMPOL.

Table 20.1. Comparison of SBL key words to SIMPOL equivalents

SBL	SIMPOL	Comments
AND	**and**, **AND**	The **AND** operator in SBL although not described as such in the on-line documentation is actually a bit-field operator. The reason that this is not obvious is that in most cases it is used for Boolean comparisons together with the **IF** statement and that particular statement in SBL compares with false, which is the value zero. Anything that is not equal to zero is considered to be true. In SIMPOL there are two different operators, the **and** and the **AND**. The lowercase version is used for Boolean comparisons where the result will be one of the special values: `.true` or `.false`. The uppercase version is specifically used for testing whether certain bits in a value are on or not by using a mask. For a proper explanation of bitwise operators see the Appendix in the SIMPOL Language Reference manual.
DIM, **GLOBAL**, **REDIM**, **ERASE**, **CLEAR**	typename variablename	There has been some discussion about adding the **dim** and **as** key words to the language as aliases but the current assessment is that with the advent of so many languages that use the same approach as SIMPOL there is no real advantage to providing an alternative method of declaring variables. In SBL there is a number of ways to create a variable: using the **DIM** key word within a procedure or function creates a local variable, using it outside (assuming the program is not started with a **SUB main()** creates a global variable. Forms that have variables on them cause those variables to be creat-

SBL	SIMPOL	Comments
		ed as global variables. Using the **GLOBAL** key word creates a global variable. Just using a variable name (this does not apply to object variables) within a procedure or function creates the variable locally if no variable exists with that name at the global level. If the program is not in a procedure or function and was started without a **SUB main()** then using a variable name creates a global variable. The **ERASE** command erases a single variable or multiple variables (both local and global)when using wild cards. The **ERASE** command works only if the variable is not used on a form (and never has been) if any form is open. The **REDIM** command is used to resize an array variable in SBL In SIMPOL there are no global variables, so all of that complexity disappears. All variables must be declared in SIMPOL before they can be used. If a variable is declared at one point in the function and then redeclared in another point, then it is destroyed and recreated at that point. Many SBL programmers use the **DIM** command only to create arrays. One of the common causes of difficult to detect side-effects in SBL programs is that of not dimensioning (declaring) variables in the appropriate locations. Although it allows faster programming it results in more expensive maintenance. Arrays in SIMPOL are quite different from those in most languages, since they do not need to be sized at the beginning and they are not an array of a specific type, they are themselves an object and can contain any arrangement of items desired and can also contain a mixture of types. This makes them quite flexible, but requires some thought at times to decide if they are the best approach to a problem. SIMPOL comes with a library of various pre-designed types that often provide a better solution to storing a collection of items, such as the objset and the list types. Just as the **ERASE** command is unnecessary, the same is true of the **CLEAR**.
FOR … **NEXT [STEP]**	**while** … **end while**	In addition to the **WHILE** loop construct SBL also provides a **FOR** block statement. In SIMPOL the **while** … **end while** block statement is the only looping construction. The reasoning behind this decision was that the **FOR** statement is essentially a special case of the **WHILE** and therefore unnecessary. There would be no speed advantage since the language is compiled.
FUNCTION … **END FUNCTION**	**function** … **end function**	A function in SBL is required to have a data type extension of either the dollar sign ($) or one of the numeric value symbols (%, %%, &%, #%, or !%). The return value of the function is assigned to a local variable that carries the same name as the function itself. In SIMPOL, the return value of the function is the value of the expression that immediately follows the **end function** statement. This value (or object) can be of any type and there is no standard syntactic way of telling the type of the return value of a function. The type of the return value can even change depending on certain things, such as the data typea that are passed to the function to begin with! Also, it is not required to make use of the return value in SIMPOL, so a function that has a return value can be called without assigning the return value.
IF … **THEN|GOTO** … **ELSE IF** …	**if** … **else if** … **else** … **end if**	In SBL there are several kinds of **IF** statement. There is the **IF**…**GOTO** single-line statement that has no equivalent in SIMPOL (GOTO is not supported in SIMPOL). There is also the nor-

SBL	SIMPOL	Comments
THEN … **ELSE** … **END IF**		mal single-line **IF** … **THEN** … **ELSE** command that does not require an associated **END IF** statement. Finally there is the multiline block version that requires an **END IF** statement. In SIMPOL the only form that exists is the latter block form that requires the **end if** statement. This is part of the design philosophy of SIMPOL, in that every command and/or block statement has a single entrance and exit. Also, SBL is essentially a line-oriented language that is interpreted, whereas SIMPOL is a statement-based language that is compiled. The end-of-line character still ends a statement in SIMPOL and there is also a line continuation character so that long programming lines can be spread over multiple lines. Even if an **if**-statement is on a single line in SIMPOL it must be followed by an end-of-statement character (: or ;) and then the **end if** statement.
NOT	**not**	The **NOT** operator in SBL although not described as such in the online documentation is actually a bit-field operator. The reason that this is not obvious is that in most cases it is used for Boolean comparisons together with the **IF** statement or in a **WHILE** loop as an exit condition. To work as a Boolean operator it needs to be applied to an expression and that particular statement in SBL compares with false, which is the value zero. Anything that is not equal to zero is considered to be true. In SIMPOL there are two different operators, the **and** and the **AND**. The lowercase version is used for Boolean comparisons where the result will be one of the special values: `.true` or `.false`. The uppercase version is specifically used for testing whether certain bits in a value are on or not by using a mask. For a proper explanation of bitwise operators see the Appendix in the SIMPOL Language Reference manual.
OR	**or**, **OR**	The **OR** operator in SBL although not described as such in the online documentation is actually a bit-field operator. The reason that this is not obvious is that in most cases it is used for Boolean comparisons together with the **IF** statement and that particular statement in SBL compares with false, which is the value zero. Anything that is not equal to zero is considered to be true. In SIMPOL there are two different operators, the **or** and the **OR**. The lowercase version is used for Boolean comparisons where the result will be one of the special values: `.true` or `.false`. The uppercase version is specifically used for setting certain bits in a value to the on or off by using a mask. For a proper explanation of bitwise operators see the Appendix in the SIMPOL Language Reference manual.
SELECT CASE … **CASE** … **CASE ELSE** … **END CASE | SELECT**	**if** … **else if** … **else** … **end if**	There is currently no **SELECT CASE** block statement in SIMPOL. Although the SBL block statement provides a certain ease of reading and expression in the code, it was decided that unless the implementation of a block statement of this nature actually provided more or different functionality to that of the **if** statement, it was not worth crowding the field of key words with yet another. There is discussion about adding a statement like this but with the added capability that is found in the C programming language of being able to fall through to the next case unless a **break** statement is encountered. This would add a useful capability that is not otherwise provided by the **if** statement.

SBL	SIMPOL	Comments
SUB … **END SUB**	**function** … **end function**	In SIMPOL there is no difference between a function and a procedure (**SUB**) except that a function that is used like a procedure has no return value. The other basic difference is that there is no practical limit to the number of parameters that can be passed (in SBL this is limited to 15), and parameters can be passed by name or even left out. There is no **CALL** key word in SIMPOL, functions are called by using their name directly.
WHILE … **WEND**	**while** … **end while**	The SBL version of the **WHILE** loop always tests the condition at the beginning of the loop. It also allows the programmer to break out of the loop using the **END WHILE** command. In some cases programmers have used additional **WEND** statements inside the loop to cause the program to immediately return to the beginning of the loop, but this is technically incorrect and is not supported by the language. The SIMPOL version allows a condition at the beginning and the end of the loop, and either or both can be set. There is no command for breaking out of the loop from somewhere in the middle, in keeping with the design philosopy of SIMPOL. Some languages provide a slight variation of the **WHILE** loop known as: **REPEAT** … **UNTIL** or **Do** … **Loop While**. This block statement allows the block to always execute once before the test is applied since the test is at the end of the block. In SIMPOL this is accomplished by using the **while** statement with an ending condition but no starting condition. In current SIMPOL code it is quite common to see both conditions used: the first for the main test and the last to test for errors. The final test is best read as: "end the while if the condition is true".

SBL Commands and Functions and the SIMPOL Equivalents

The following table contains an alphabetical list of SBL key words and their SIMPOL equivalent together with some explanatory text describing the differences.

Table 20.2. Comparison of SBL commands and functions to SIMPOL equivalents

SBL	SIMPOL	Comments
`ASC()`	`.charval()`	The `ASC()` function in SBL returns the ASCII (OEM) value of the first character in the string that is passed as the argument. In SIMPOL the `.charval()` function returns the Unicode character value of the first character in the string passed as the argument.
BLANK	type(db1table).`newrecord()`	Creating a new record in a database in SIMPOL is done by calling the `newrecord()` of the associated database table object. Unlike with SBL, the default formulae are not executed at the point in time of creating a new record, so it is the re-

SBL	SIMPOL	Comments
		sponsibility of the SIMPOL programmer to perform any default calculations and assign the results to the associated fields.
CALL	`!execute()`	The **CALL** command in SBL that is used to execute external programs has its equivalent in SIMPOL in the form of the `!execute()` system function. One current difference between the two is that the SIMPOL version does not create a shell, so if you are using it in Windows to call things like the **COPY** or **DEL** commands, you need to call the command shell with appropriate command line switches or call a batch file that contains the commands instead.
`CHAR$()`	`.char()`	The `CHAR$()` function in SBL returns the value passed as an ASCII (OEM) character. In SIMPOL the `.char()` function returns a Unicode character that is the equivalent of the value passed as the argument.
`DATE$()`	`DATESTR()`	The `DATE$()` function in SBL takes a date and an optional format string and returns the date as a string formatted using the format string passed. The SIMPOL version requires the format string to be passed. Supported formats are the same in both versions: "day month year", "month day year", or "year month day". Separators can be any character, though sensible choices should be made. The actual format string supports the following:

Table 20.3.

dd	Day no leading zero
0d	Day with leading zero
zd	Day with leading space
mm	Month no leading zero
0m	Month with leading zero
zm	Month with leading space
mmm	Three letter abbreviated month name
mmmm	Month fully spelled out
yy	Two digit year

SBL	SIMPOL	Comments
		yyyy — Four digit year
DAYS	`string2date()`	The SBL command **DAYS** takes either a date or a text containing a date and returns an integer representing the number of days since 01 January, 0001. The SIMPOL version only supports converting a date expressed as a string. It requires a format string in order to know how to process the date. It returns a date object represnting that time. The value of a date object is an integer containing the number of days since 01 January, 0001. **Note** Because of an error in the way SBL calculates the dates prior to the Gregorian Reform in England (September 2, 1752), the value of the days in SBL is 11 days off. Also, SIMPOL starts from 0, rather than 1, so the effect is actually a difference of 10 days. This normally makes no difference, but can become an issue if working with actual integer values and using the data in both SBL and SIMPOL as a hybrid system. Also, in SIMPOL no support is provided for the Gergorian Reform. Instead the Julian integer value for the number of days assumes no error occurred. For historical dates it would be necessary use your own date formatting function as this is considered a localization issue.
END	no equivalent	The SBL command **END** allows the program to stop executing. In SIMPOL programs will exit only when the reach an error condition or they exit through the end of the `main()` function.
`ERR$()`, **ERRNO**, **ERROR**,	No equivalent	In SIMPOL there is no error handling in the form of interrupts such as is the case

SBL	SIMPOL	Comments
ON ERROR, **RESUME**, etc.		in SBL. Instead, most function calls that can cause an error take an error object and in some cases an error text object. In the case of an error, if the error object has been passed, then the error will be returned in the object. If no object has been passed, then the program will halt at that point with an error. Most syntax errors will be found during compilation and post-processing of the IDE. In some cases errors will occur at runtime but should normally be found during testing.
`EXISTS()`	`fileexists()`	In SBL the `EXISTS()` function has two variants. One variant checks whether the argument passed exists as a file in the file system. That functionality is provided for in SIMPOL by the `fileexists()` function. The other variant takes a value and an index and returns whether a record exists with that value in the target database table without changing the current record pointer in the target index. No exact equivalent exists for this since none is really needed. There is a function called `lookup()` that is found in the `appframework.sml` library and which takes an index object, a value, and an error variable and which returns a record object if a match is found, otherwise it returns `.nul`.
`FCASE$()`	`.tcase()`	In SBL to convert a string such that only the first character is capitalized the programmer calls the `FCASE$()` function; the equivalent in SIMPOL is the `.tcase()` (titlecase) function.
`FIX()`	`.fix()`	The `FIX()` function is commonly used in SBL to ensure that a floating point value is as close as possible to a desired number of decimal places as desired (floating point numbers are not precise because base ten fractions are not reliably representable in binary). In SIMPOL the more important use of the `.fix()` function is to truncate the exactly precise but potentially extremely large number of trailing digits from a value. It would not be uncommon to have a decimal value as the result of a division operation that had tens, hun-

SBL	SIMPOL	Comments
		dreds, or even thousands of digits trailing the decimal point.
`HRS()`	`HRS()`	There is essentially no difference between these two functions, other than that in SIMPOL the parameter passed must be a time object. In both cases the number of hours in the time are returned as an integer.
`IF()`	`.if()`	There is essentially no difference between these two functions, other than that in SIMPOL the argument must result in a Boolean value of either `.true` or `.false`, whereas in SBL zero is false and non-zero is considered to be true.
`INSTR()`	`.instr()`	In both SBL. and SIMPOL these functions are used to determine whether some substring can be found in the target string. The only real difference between the two is that the SBL version can take an optional leading parameter that tells the function where in the string to begin looking. The equivalent in SIMPOL is to pass only the portion of the string in which to look, and then to adjust the value returned by adding the offset to the beginning of the substring that was passed.
`IS()`	`=@=`	In SBL the `IS()` fuction is used to compare if two variables refer to the same object. In SIMPOL this handled using an operator. This question can be negated in SBL by applying the **NOT** to the result of the function. In SIMPOL there are two equivalent operators for this: **!@=** and **<@>**.
`LCASE$()`	`.lcase()`	In SBL to convert a string to lowercase the programmer calls the `LCASE$()` function and in SIMPOL the `.lcase()` function serves the same purpose.
`LEFT$()`	`.lstr()`	These two functions are essentially identical in their function: they return the portion of the string from the first character until the end of the string or until the position passed whichever is less.
`LEN()`	`.len()`	Both in SBL and SIMPOL these functions return the length of the argument

SBL	SIMPOL	Comments
		passed. One difference is that in SIMPOL this is the length of the argument in characters that are Unicode characters, whereas in SBL these are single-byte ASCII (OEM) characters.
LIKE	`.like1()`	For the most part the two versions of **LIKE** work the same. There are a few more options in the SIMPOL version, such as optional case-sensitivity, but otherwise they should be compatible (other than the fact that the SBL version is an operator and the other is a function).
LOAD	`!loadmodulefile()`	The SBL **LOAD** command is used to load program files into memory and is most commonly used with the **, NEW** option to load a set of routines into memory for use by the program. It is also used to load queries, updates, text editor files, function key files, and labels definitions. Almost all of these latter items are better dealt with in SIMPOL as methods of the associated object. The function `!loadmodulefile()` is a SIMPOL system function for loading a compiled SIMPOL library so that its exported types and functions can be used. Although it is possible to directly include a library module in the resulting program when the program is compiled, it may be more efficient in some cases to load the module as needed, for example when the module may not always be needed.
`LOCK()`	ppcstype1file.locked, ppcstype1record.locked, sbme1table.locktype, sbme1record.locktype	In SBL the `LOCK()` serves to test whether a given record is locked in a database file. A similar capability exists in SIMPOL except that what is tested is the value of a property of the file (table) or record object. One difference in this is that this will only tell if the user has locked the record or table, it will not tell if others have done so (or even if another object in the same program has done so).
LOCK ALL	ppcstype1file.`lock()`, sbme1file.`lock()`	These two items are very similar, other than from an architectural perspective: with one being a command and the others being methods of types. In all cases the file is locked. In the case of the sbme1 type, it is also necessary to hold

SBL	SIMPOL	Comments
		at least a shared lock on the table in order to create records.
`MAX`	`.max()`	The SIMPOL version of this function simply takes an unlimited number of arguments and returns the one that is of the highest value. The SBL version can only be used with arrays or on reports under special circumstances.
`MID$()`	`.substr()`	These two functions are virtually identical, except that in SBL to return everything until the end of the string, the last parameter is left out, whereas in SIMPOL all three parameters are always required so to return everything the last parameter can be set to `.inf`.
`MIN`	`.min()`	The SIMPOL version of this function simply takes an unlimited number of arguments and returns the one that is of the lowest value. The SBL version can only be used with arrays or on reports under special circumstances.
`MINS()`	`MINS()`	There is essentially no difference between these two functions, other than that in SIMPOL the parameter passed must be a time object. In both cases the number of minutes in the time are returned as an integer.
MOD	**mod**	These two operators do the same thing, they return the fractional portion of a division operation.
`MOD()`	no equivalent	The `MOD()` function in SBL is intended to indicate whether the current record in the file passed as the argument has been modified. Since there is no such thing as a current record (current file, etc.) in SIMPOL this function is meaningless. At some point when data-aware forms have been added there may be a method to indicate if any record on the form has been modified since it was read. That would be the appropriate location for such functionality.
NOT	**not**	These two operators are essentially the same, other than that the SBL version operates with `0` and non-`0` and the SIMPOL version works with `.false` and `.true`.
`NOTHING`	`.nul`	The literal value `NOTHING` in SBL is used exclusively together with the `IS()`

SBL	SIMPOL	Comments
		to test whether an object variable refers to nothing. In SIMPOL this test can be carried out using the **=@=** operator and the literal value `.nul`. This value is used in many different areas and ways within SIMPOL.
QUIT	no equivalent	The SBL command **QUIT** allows the program to suddenly exit, closing down the Superbase environment after calling the OnUnload event procedure of the Superbase object (if it was set). SIMPOL programs are self-sufficient so there is no additional environment to shut down and they will exit only when the reach an error condition or they exit through the end of the `main()` function assuming that all threads have also ended.
REM, '	', ", //	The **REM** statement and the single quote character can both be used to indicate a comment in an SBL program. The single quote character can also immediately follow a command in SBL. In SIMPOL both the single and double-quote characters can be used to indicate a comment but unlike SBL, in SIMPOL these are only considered to be comment characters if they are on the left side of an equation (at the beginning of a statement). Also, if another matching quote is found inside, then the comment is ended and must be followed by an end-of-line character or end of statement character (: or ;). Using this technique a comment can be embedded in the middle of a line of code. The only line-level comment is the double forward slash (//). This must be placed at the beginning of a statement (either at the beginning of a line or directly following an end of statement character and separated only by white space).
`REPLICATE (svar$, nvar%%)`	**nvar * svar**	The `REPLICATE()` is part of the standard BASIC repertoire and SBL includes this function to replicate a string a given number of times. This function is unnecessary in SIMPOL since it is possible to directly multiply a string by an integer and thereby replicate the string that many times.

SBL	SIMPOL	Comments
`RIGHT$()`	`.rstr()`	These two functions are essentially identical in their function: they return the portion of the string from the last character until the beginning of the string or until the number of characters backwards from the end of the string, whichever is less.
`SECS()`	`SECS()`	There is essentially no difference between these two functions, other than that in SIMPOL the parameter passed must be a time object. In both cases the number of seconds in the time are returned as an integer.
SELECT FIRST INDEX ""	**db1tablevar.select(lastrecord=.false, error=e)**	One of the significant differences between SBL and SIMPOL is the fact that in SBL, the entire Superbase product is always present, and there is always a globally visible current database table (or file), for each database table (file) there is a current index, and each index has a current record that may be different for each index. There is also a currently loaded record. When working with multiple windows open, this gets even messier still, since each `ViewWindow` may have different database tables open, or even the same ones but with a different set of current indexes and records. This must be carefully managed using the `SetSBLWindow()` method of the Superbase object. In SIMPOL there are no global variables. To access a record from a database table a method of either the table, an index of the table, or even a record of the table is called. There is no current table, no current index, and no current record. Although the dataform1 type provides for a current master record for a data-aware form, this does interfere elsewhere in the system. A record object is the result of calling some form of select method. There can be as many record objects as the programmer wishes to create. They can be stored in lists or arrays. The logic behind how the select methods are designed is as follows: • table.select() — Tables know what is at the beginning and the end of the sequential order of the table.

SBL	SIMPOL	Comments
		• index.select() — Indexes know what is at the beginning and the end of the index order. • index.selectkey() — Indexes know how to find a value in the index using a key. • record.select() — Records know where they are in whatever method selected them, and can get to the previous and next items in the same selection order, so if they were selected using an index, they can find the previous and next records in the same index, if selected using the sequential order of the table, they can find the previous and next items in the sequential order of the table. The return value of a record selection in SIMPOL is a record object. If an error occurs, then the record object may be equal to `.nul`. Always pass an integer object to these methods to trap any error that occurs. The integer must be pre-initialized to `0`, since the error variable will only be written to if an error occurs. If the variable is not set to `0`, then the program may incorrectly assume that a pre-existing value was returned by the call to the method. Specific to this command, by using the double-quote `""` argument for the *`INDEX`* parameter, Superbase is being told to select the first record in the sequential order of the table. Using the `select()` method of table object, SIMPOL is doing the same thing.
SELECT LAST INDEX ""	**db1tablevar.select(lastrecord=.true, error=e)**	See the prior entry for **SELECT FIRST INDEX `""`** for details about how to select records using SIMPOL. In this particular case, by using the double-quote `""` argument for the *`INDEX`* parameter, Superbase is being told to select the last record in the sequential order of the table. Using the `select()` method of the table object, SIMPOL is doing the same thing.
SELECT FIRST INDEX RecNo.TEST	**db1indexvar.select(lastrecord=.false, error=e)**	See the prior entry for **SELECT FIRST INDEX `""`** for details about how to select records using SIMPOL. Here, the

SBL	SIMPOL	Comments
		RecNo index is being used as an argument to select the first entry in that index. In SIMPOL this can be done using a variable that refers to the RecNo index, or it may be done using valid object syntax to reach the index object. For example: **db1tablevar!RecNo.index.select(lastrecord=.false)** uses the table variable. From the `db1tablevar` variable the member operator (`!`) is used to retrieve the field object for the `RecNo` field, and then its index property is accessed using the dot (`.`) operator, and again using the dot (`.`) operator, the `select()` method is called.
SELECT LAST INDEX RecNo.TEST	**db1indexvar.select(lastrecord=.true, error=e)**	See the prior entry for **SELECT FIRST INDEX ""** for details about how to select records using SIMPOL. See the prior entry for **SELECT FIRST INDEX RecNo.TEST** for details about how to select records using an index in SIMPOL. The only difference to the **SELECT FIRST** version is that the *`lastrecord`* parameter is assigned the value `.true` rather than the value `.false`.
SELECT KEY 123 INDEX RecNo.TEST	**db1indexvar.selectkey(123, error=e, found=f)**	See the prior entry for **SELECT FIRST INDEX ""** for details about how to select records using SIMPOL. See the prior entry for **SELECT FIRST INDEX RecNo.TEST** for details about how to select records using an index in SIMPOL. In this particular case, the `selectkey()` method is being used. The value that is being looked up must match the data type of the field for which the index was created. The only variation of that is that an integer value can be used to search within indexes on date, time, and datetime fields. If the record is successfully found, then the boolean variable (which must be pre-initialized) that was passed to the *`found`* parameter is set to `.true` and the variable passed to the *`error`* parameter will be unchanged. If the *`found`* parameter is not passed, and the record is not found, then the return value will be `.nul` and an error value will be as-

SBL	SIMPOL	Comments
		signed to the variable that was passed to the `error` parameter.
SELECT NEXT	**db1recvar.select(previousrecord=.false, error=e)**	See the prior entry for **SELECT FIRST INDEX ""** for details about how to select records using SIMPOL. As stated in that entry, the return value of a selection is a record object. To select the next record (or the previous one) the `select()` method of the record object is called, passing the appropriate value to the *`previousrecord`* parameter, in this case the value `.false`.
SELECT PREVIOUS	**db1recvar.select(previousrecord=.true, error=e)**	See the prior entry for **SELECT FIRST INDEX ""** for details about how to select records using SIMPOL. As stated in that entry, the return value of a selection is a record object. To select the next record (or the previous one) the `select()` method of the record object is called, passing the appropriate value to the *`previousrecord`* parameter, in this case the value `.true`.
SET INDEX Name.TEST	**db1recvar.selectcurrent(db1indexvar_Name, error=e)**	This command is supplied by Superbase to allow the programmer to change the controlling index of an already selected record. In SIMPOL, it is necessary to call the `selectcurrent()` method and to pass the desired index object to switch to a different controlling index. If no index parameter is passed, then the default is to use the value `.nul`, which results in the record being switched to having been selected using the sequential order of the table. It is important to remember this when reselecting a record with a lock, since otherwise the record may be switched away from the desired index without realizing it!
SELECT FIRST LOCK INDEX ""	**db1tablevar.select(lastrecord=.false, lock=.true, error=e)**	See the prior entry for **SELECT FIRST INDEX ""** for details about how to select records using SIMPOL. The only significant difference here is that in both cases the relevant locking parameter *`LOCK`*" or *`lock`* is being passed. In SBL if the locking operation fails, then an error occurs which may result in a call to a global error handler, or if the error has been disabled, then it will simply set the value of the `ERRNO` system value. In SIMPOL this will result in a return value of `.nul`, and the variable

SBL	SIMPOL	Comments
		passed in the *error* parameter will be set to the error value that was the cause of the problem.
SELECT FIRST LOCK INDEX RecNo.TEST	**db1indexvar.select(lastrecord=.false, lock=.true, error=e)**	See the prior entry for **SELECT FIRST INDEX ""** for details about how to select records using SIMPOL. See the prior entry for **SELECT FIRST LOCK INDEX ""** for details about how to select records with a lock using SIMPOL.
SELECT KEY 123 LOCK INDEX RecNo.TEST	**db1indexvar.selectkey(123, lock=.true, error=e)**	See the prior entry for **SELECT FIRST INDEX ""** for details about how to select records using SIMPOL. In SBL it is a risky venture to use the *LOCK* together with a **SELECT KEY** statement, since if the selection fails to find the correct record, it will still find a record and will lock that one instead. It is better practice to make sure the record has been found and then use the **SELECT CURRENT LOCK** command to lock the record. The same is also true of SIMPOL, though it is possible to do this safely, simply by not passing a *found* parameter.
SELECT NEXT LOCK	**db1recvar.select(previousrecord=.false, lock=.true, error=e)**	See the prior entry for **SELECT FIRST INDEX ""** for details about how to select records using SIMPOL. See the prior entry for **SELECT FIRST LOCK INDEX ""** for details about how to select records with a lock using SIMPOL.
SELECT CURRENT LOCK	**db1recvar.selectcurrent(lock=.true, error=e)**	See the prior entry for **SELECT FIRST INDEX ""** for details about how to select records using SIMPOL. See the prior entry for **SELECT FIRST LOCK INDEX ""** for details about how to select records with a lock using SIMPOL. When selecting the current record in SIMPOL it is important to make sure that the *index* parameter is assigned an appropriate value since otherwise it will default to `.nul` and potentially change the current index (though only of the record that is returned). To retain the same index that was used in the original selection, it is easiest to just pass the index property of the record object, such as: **db1recvar.selectcurrent(db1recvar.index, lock=.true, error=e)**.

SBL	SIMPOL	Comments
SELECT RE-MOVE	**db1recvar.delete(error=e)**	In SBL, once the record is deleted it is simply gone. In SIMPOL, the record object still exists and can be treated like a new record object that is not yet stored. This means that a record could be deleted and the record object could then be used (perhaps modified) to create a new record, and then that record could be saved.
`SPACE$ (nvar%%)`	**nvar** `* " "`	Probably related to its BASIC heritage, SBL includes this function to create a string a given number of space characters in length. This function is unnecessary in SIMPOL since it is possible to multiply a string by an integer and thereby replicate the string that many times.
`STR$()`	`.tostr()` or `STR()`	The `STR$()` function in SBL allows a large number of different methods for formatting the number as a string. The equivalent function in SIMPOL simply formulates the number as a string and also requires the base to be provided. When used for base ten numbers, it is roughly equivalent to the command **STR$(nvar%%, ".")**, except that in the case of a zero value the character `0` will be output whereas in SBL the empty string is the result. For a version that is directly compatible with the SBL version (except for the lack of support for scientific notation), look for the `STR.sml` library. It is also provided in source code. One difference between these is that the SIMPOL library function requires the user to provide an object that includes the numeric settings for decimal point, thousands separator, currency symbol, and whether the currency symbol is a prefix or suffix. This is necessary since there are no such global settings in SIMPOL.
`THOUSECS()`	`THOUSECS()`	There is essentially no difference between these two functions, other than that in SIMPOL the parameter passed must be a time object. In both cases the number of thousandths of a second in the time are returned as an integer.
`TIME$()`	`TIMESTR()`	The `TIME$()` function in SBL takes a time value and an optional format string

SBL	SIMPOL	Comments
		and returns the time as a string formatted using the format string passed. The SIMPOL version requires the format string to be passed. Supported formats are the same in both versions. Separators can be any character, though sensible choices should be made. The actual format string supports the following:

Table 20.4.

hh	Hours
mm	Minutes
ss	Seconds
.s	Thousandths of a second
am	12 hour clock

Some typical examples of time format strings for a time of 1:35 pm might be:

Table 20.5.

hh:mm	13:35
hh:mm:ss	13:35:00
hh:mm am	1:35 pm
hh:mm:ss.s	13:35:00.000

SBL	SIMPOL	Comments
`TIMEVAL()`	`string2time()`	In SBL the `TIMEVAL()` function takes either a time or a string representation of a time. The SIMPOL `string2time()` function only accepts a string and a format string and it returns a time object.
`UCASE$()`	`.ucase()`	In SBL to convert a string to uppercase the programmer calls the `UCASE $()` function and in SIMPOL the `.ucase()` function serves the same purpose.
`VAL()`	`.toval()`	The `VAL()` function in SBL is used to convert a string to a number. It is somewhat idiosyncratic in the way that it works. All leading whitespace is ignored, as are currency symbols and thousands separators and the number is returned that is found up until the first non digit character following the first decimal point or the end of the string is reached. The SIMPOL version of this in keeping with its support for multi-

SBL	SIMPOL	Comments
		ple bases, takes the value, the characters to ignore, and the base to use for interpreting the string as a number. It might seem a bit awkward dealing with defining the characters to ignore in SIMPOL since it could be all of the Unicode character set, but in actuality it is quite easy, since it is also possible to subtract strings from strings in SIMPOL. To define the set of characters to ignore, simply subtract each of the characters that are desired from the string being evaluated, like this: **n = .toval(s, s - "0" - "1" - "2" - "3" - "4" - "5" - "6" - "7" - "8" - "9" - ".", 10)** , which will result in all of the desired characters being removed from the string and all of the remaining characters being ignored. For a more typical SBL version check the library for the `VAL.sml`.
WAIT FOR nvar %%	`!wait()`	These both wait for a specified amount of time. Only the duration and intervals are different.

Differences Between SIMPOL and SBL

Things that you could skip in SBL are required in SIMPOL, such as declaring variables and initializing them. Although that means a bit more work to get something going, the IDE is designed to make using SIMPOL as easy as possible. Also, there are no global variables in SIMPOL, but it is quite straightforward to create a type that contains all of the quasi-global information, initialize that type during the start of the program, and then just pass that type around everywhere the program needs it. The event objects in SIMPOL are specifically designed to allow the optional assignment of a reference to any type of object, and that object is then passed to the event handling function as a parameter.

Tools for Converting SBL to SIMPOL

There are a number of conversion tools supplied with SIMPOL Professional to convert various aspects of Superbase packages. Some are written in SBL and some are written in SIMPOL. Here is a list of them:

- `sbf2sbm.smp` – Converts Superbase `SBF` files to SIMPOL `sbm` format (files should be unencrypted, reorganized, and preferably without passwords)
- `sbv2xml.sbp` – SBL program to convert Superbase forms to SIMPOL XML forms
- `sbvr2xml.sbp` – SBL program to convert Superbase graphical reports into SIMPOL XML format graphical reports (does not convert perfectly, some adjustment of the results will be needed, primarily paper size and calculations)
- `ngmengen.sbp` – SBL program that converts a Superbase menu program into a SIMPOL source code file

- `dlg2sma.sbp` – SBL program that converts Superbase dialogs programs (as saved from the designer) into SIMPOL source code
- `sbd_formula_reader.smp` – Reads Superbase `SBD` files and creates a source code file that contains a function for constants, calculations, and validations for each Superbase file read where such formulae are in use. The resulting code will need to be hand-edited to be usable, since a complete formula conversion tool is not included.

The best initial approach is to convert the database tables and the forms, if there is a menu program available, convert that, grab a copy of the Address Book example and make a new project from it. Use the first form as an initial step and get it coming up using the Address Book code, modified as required. Once that is happening, use the appwindow.`openformdirect()` method to open the next form into the same window as a response to a menu event. From there convert any formulae using the `sbd_formula_reader.smp` converter and then hand adjust the resulting source code. Remember, there is an equivalent for `LOOKUP()` in SBL called `lookup()` and it can be found in the `uisyshelp.sml` library.

From this point onward, it just depends on how the application is constructed. Fill in all the functions and the navigational structure. Use the tool bar from the Address Book sample, or leave it out, or design your own.

To assign calculated values and do validations of field content before a record is saved, assign an event handling function to the dataform1.onsave event. The return value should be `.true` if the record should be saved, and `.false` if there is a problem and the record should not be saved. For further information about working with the Application Framework, see Chapter 26, *Using the SIMPOL Application Framework*.

Part IX. Supplied SIMPOL-Language Libraries

This part discusses the various libraries written in SIMPOL that are supplied with the language. Many of these are supplied as source code, thus providing the user with the ability to understand how the libraries work, as well as providing the ability to improve and extend them should some required functionality be missing or existing functionality be faulty.

Table of Contents

Chapter 21. SIMPOL Language Libraries Included

Introduction

One of the more powerful features of the SIMPOL programming language is the ability to produce libraries of reusable functions and types. Part of the underlying design philosophy in SIMPOL has been to produce as much as possible using the language and to use the C programming language for implementing core language and heavy-use components, and for improving the speed of SIMPOL-based code when those areas are clearly identified as requiring such improvement.

In this chapter we will discuss briefly the supplied SIMPOL language libraries. Since the libraries themselves occasionally go through revisions it is a good idea to regularly look into the source directory for those libraries supplied as source (most of them) and see what is new or has changed. It is also recommended to look at the source code to the various libraries for options, function parameters, and also to examine how the functions and types are written. If you are having a problem with a library type or function that you have the source for, you can use that source to debug your program. If there is an error in the library, please let us know.

List of Supplied Libraries

The following table contains a list of the supplied libraries, an X if they are supplied as source, and a brief description of each one. Some of the libraries may be more fully explained in a separate section below.

Table 21.1. Supplied SIMPOL-Language Libraries

Name	Source	Description
`abs.sml`	X	Implements the `ABS()` function for compatibility with SBL. It can also be used in general in SIMPOL, since there is no equivalent.
`appframework.sml`	X	Implements an application framework for working with data-aware form style applications. This framework is used by the samples supplied with the SIMPOL Quickstart Guide.
`boolstr.sml`	X	Provides functions for converting from and to boolean and datetime types to strings.
`bzip2.sml`	X	Wrapper for the BZip2 compression library.
`calceval.sml`	X	Contains the `calceval()` function for evaluating a formula contained in a string and returning the result.
`codepageslib.sml`	X	Provides functions for converting from and to SIMPOL characters for various code pages.
`colorpalette.sml`	X	Supplies types and functions for working with colors and palette entries. This is primarily used by `imagelib.sml` for saving images to disk.
`conflib.sml`	X	Provides functions reading from and for writing to configuration files that follow the standard for INI files in Win-

Name	Source	Description
		dows. In the future other configuration file formats may be supported in this library.
`databaseforms.sml`	X	This library implements data-aware, multi-page forms. It contains the entire set of types from the dataform1 family. For more information on programming with these types, see Chapter 23, *Using Data-Aware Forms in SIMPOL*.
`datetimelib.sml`	X	This library provides several date, time, and datetime functions and includes other related libraries to provide a single library for inclusion.
`db1lib.sml`	X	Implementation of a dummy group of classes based on the db1 type tags. The purpose is to ensure that viable inline help in the IDE is provided for variables declared using the db1 type tags.
`db1util.sml`	X	Provides numerous functions for working with databases. Functions for copying one record to another, determining whether a field is valid or is indexed, converting from field values to string and the reverse, and other functions.
`dbconverter.sml`		This is the primary data conversion library, for both import and export converters. The design uses a common record structure that is supported by import and export converters as the medium of exchange. Any import converter can be hooked up to any export converter.
`errormsgs_en.sml`	X	This error messages library provides a standard method of returning a consistent error message for any of the standard error values listed in `errors.sma`. This library implements the English language messages.
`fastset.sml`	X	This implements the fastset data type for working with sets that allow for string-indexing of objects. This library should be in preference to the objset library in newer code. It is virtually identical in its API but is considerably faster in execution. When working with sets of values or sets of objects that do not need to be string-indexed, it is even faster to use the built-in set data type.
`filesyslib.sml`	X	Provides functions for working with files and directories. Currently this includes a function to retrieve the correct directory separator character and another to parse file and path names into their component parts.
`formlib.sml`	X	This library provides the additional functionality for loading and saving dataform1 and printform1forms. It also provides the functionality to save dataform1 forms as program source code and to save them as a wxform source program with all the data aware aspects stripped away.
`gaugelib.sml`	X	This library includes two types for providing a gauge dialog that can be shown and updated in order to inform the user while your program is doing long operations.
`graphicreportlib.sml`		This library provides the Graphic Report functionality including saving and loading these reports.

Name	Source	Description
`httpclientlib.sml`	X	Contains functions for accessing web pages on the Internet. Includes functions for both the `GET` and `POST` style access of web pages.
`imagelib.sml`	X	Provides numerous functions for working with databases. Functions for copying one record to another, determining whether a field is valid or is indexed, converting from field values to string and the reverse, and other functions.
`int.sml`	X	Implements the `INT()` function for compatibility with SBL. It can also be used in general in SIMPOL, since there is no equivalent.
`jpeglib.sml`	X	Provides types and functions for working with JPEG files. Currently the only functionality is the function to retrieve the size of a JPEG image and a wrapper function to allow that function to be called via the SMEXEC32.DLL interface.
`libxml.sml`		Provides the full implementation of the XML Document Object Model (DOM) Core Level 1 and Level 2 with some additional capabilities from Level 3. It provides as well, the ability to do XSLT transforms, document validation, and support for HTML documents. There is an example program in the directory `projects\dom`. That directory also contains documentation about the DOM in HTML format.
`lists.sml`	X	Various list and similar types. Includes: list, dlist, ring, queue, and stack. The dlist implementation has gone through extensive testing and modification. Most of these types are meant to be embedded into other types, to provide the ability to manage them in a list.
`ltrim.sml`	X	Implements the `LTRIM()` function for compatibility with SBL. For a more flexible implementation see the `ltrim()` function in the `stringlib.sml` library.
`mathlib.sml`	X	Contains functions for working with mathematics, such as `sin()`, `cos()`, `tan()`, `sqrt()`, and others.
`mrulib.sml`	X	This library implements a data type for managing most recently used lists, commonly shown as items on a menu. To that end, it can actually manage an entire submenu on its own, including showing a dialog for entries beyond a certain number, managing the entries in a configuration (INI) file, etc.
`netinfolib.sml`	X	This library is used for providing network-specific information, such as the currently logged-in user's name.
`objset.sml`	X	The objset type and related types are created in this library. This provides a fairly powerful and robust set implementation, including differencing, intersection, and unification of sets. The sets use a string key for sorting the entries (and for deciding if they are the same) and an optional element that is declared as `type(*)` that can contain a reference to any object.

Name	**Source**	**Description**
`odbc2.sml`		SIMPOL language support library for working with the ODBC client support.
`pad.sml`	X	Implements the `PAD()` function for compatibility with SBL. It can also be used in general in SIMPOL, since there is no equivalent.
`printformlib.sml`	X	Contains types and functions that support printing forms to the print architecture used in SIMPOL. Also implements a function for printing a record from a database.
`quickreportlib.sml`		Provides an easy to use fully functional report engine with grouping, sorting, and aggregate values at both group and report level. Reports are limited in the way they can look. For a more complex and flexible report engine, see the `graphicreportlib.sml`.
`random.sml`	X	The pseudo-random number generation provided by this library is quite useful. It uses a standard algorithm for generating pseudo-random numbers. If the seed is repeated, then the sequence will be the same each time. If a different sequence is desired, then the current date and time can be passed as the seed. The numbers generated are between 0 and 1, so any multiplier can be used to get the values and ranges desired.
`registrylib.sml`	X	The Windows registry is commonly used for storing configuration data on Windows. This library can be used to access the registry. Please be aware that user programs cannot write to the `HKEY_LOCAL_MACHINE` key on Windows Vista and later, these writes will be virtualized.
`reorglib.sml`		The functionality needed to reorganize a database container, or just individual tables within the container, including support for the system tables provided by the `db1util.sml` library, are located in this library.
`replace.sml`	X	This is a standard string replace function. It has been fairly thoroughly tested and should be able to handle situations that many string replace functions fail on, such as the replacement string or the search string being a substring of the other.
`reportlib.sml`		This library provides the core report1 type family that can be used to create custom report types. It is also embedded into the `graphicreportlib.sml` and the `quickreportlib.sml` libraries, each of which provide a specific style of report engine.
`rsalib.sml`	X	Provides a usable library for encrypting and decrypting as well as generating public and private keys.
`sbislib.sml`	X	This library supplies functions that are intended to ease the conversion of systems written for the Superbase Internet Server suite for the older Superbase product.
`sbldatelib.sml`	X	Implements the `DATESTR()`, `DAY()`, `DAYS()`, `DAYSTR()`, `MONTH()`, `MONTHSTR()`, and `YEAR()` functions for compatibility with SBL. Some of them can

Name	**Source**	**Description**
		also be used in general in SIMPOL, since in many cases there is no equivalent. This library also includes a `string2date()` function.
`sblexten.sml`	X	This library is a nearly 1:1 conversion of the Superbase library of the same name and supplies a group of useful functions, some implemented multiple times, one for each supported datatype, such as `Floor()`, `round()`, `Between()`, `Average`, and others.
`sbllib.sml`	X	This library consolidates all of the SBL-specific libraries together with a number of the FN functions, such as: `FN_Ext()`, `FN_Root()`, `FN_Alpha()`, `FN_Dec()`, etc.
`sbllocaledateinfo.sml`	X	This library provides the SBLlocaledateinfo type that is required by many of the SBL date functions. This type holds the locale information such as the names of the days of the week, the months, and the month abbreviations, plus the value for the century base for interpreting 2-digit years.
`sbltimelib.sml`	X	Implements the `TIMESTR()` and `TIMEVAL()` functions for compatibility with SBL. Both of these can also be used in general in SIMPOL, since in many cases there is no equivalent, although see also the `smtpdatelib.sml`. This library also includes a `string2time()` function.
`sbnglib.sml`	X	Implements some commonly used types, such as datasourceinfo, tbinfo, and wxformoptiongroup to provide group management of option buttons.
`sendkeys.sml`		Contains a SENDKEYS implementation that works well in Win32, including the ability to send keystrokes to a window and not just commands. Win32-only.
`sendmail.sml`	X	Provides an easy-to-use `sendmail()` that makes use of the `smtpclientlib.sml` and `smtpdatelib.sml` libraries to send simple text messages.
`serialize.sml`	X	Contains the functionality to serialize even fairly complex objects to a file such that the state of the object can be retrieved later. Obviously cannot support properties that represent data types that cannot be instantiated with the `.new()` method.
`shellexecute.sml`	X	Supplies a wrapper for the Win32 `ShellExecute()` API call, which not only will run programs, but can also be used to start the registered program for a specific file type based on its file extension, such as starting the default browser for files ending in `htm`.
`simpolpacker.sml`	X	Implements an archiving system that utilizes the BZip2 compression support from the `bzip2.sml` library, which provides single file compression.
`smtpclientlib.sml`	X	Provides basic SMTP email sending capabilities. This is a work in progress. The library does not currently support MIME, attachments, HTML email, etc. It works fine for sending straight text messages to one or more addresses.

Name	**Source**	**Description**
`smtpdatelib.sml`	X	Provides an SMTP compliant function for creating a date string from a datetime object.
`sortlib.sml`	X	Provides various sorting algorithm implementations, including Insertion Sort, Quicksort (recursive), a combination of Quicksort and Insertion Sort (even faster than Quicksort alone), and others.
`soundlib.sml`	X	Basic sound support that currently only supports the Windows operating system.
`sql1.sml`		A query engine implementation including query optimizer.
`str.sml`	X	Implements the `STR()` function for compatibility with SBL. It can also be used in general in SIMPOL, since there is no equivalent. The only thing not supported is scientific notation.
`stringlib.sml`	X	Contains various string parsing and manipulation functions. This library is heavily used in the more complex libraries. Some of the functions included are: `parsetoken()`, `ltrim()`, `rtrim()`, `multiinstr()`, `formatlinebreaks()`, etc.
`timer.sml`	X	This is a very basic but usable implementation of a timer object. Use the timer to run things that need to happen regularly independent of the rest of program execution. Each timer runs in a separate thread.
`trim.sml`	X	Implements the `TRIM()` function for compatibility with SBL. For a more flexible implementation see the `rtrim()` function in the `stringlib.sml` library.
`uisyshelp.sml`	X	Contains functions and types useful in working with the user interface and the operating system, such as retrieving the list of system colors, the default font, display size, etc. More details on this can be found in: the section called “Dialogs Using Standard Buttons”.
`unittest.sml`	X	A basic regression testing library that implements types for running regression tests and which compare the result of each test with the expected result and report only on failure.
`urlendecode.sml`	X	Provides functions for URL-encoding and URL-decoding. This is primarily used by web applications or programs that need to speak to a web server.
`urllib.sml`	X	Implements a type and function for parsing a string into a URL that has been divided into its component parts.
`utf8lib.sml`	X	Provides functions for converting to and from UTF-8 format.
`uuencode.sml`	X	Provides functions for uuencoding and uudecoding. It also has functions for doing base64 encoding and decoding. These are used by email systems for sending attachments in 7-bit characters.
`val.sml`	X	Implements the `VAL()` function for compatibility with SBL. It can also be used in general in SIMPOL, since there

Name	Source	Description
		is no direct equivalent. The only thing not supported is scientific notation.
`volatable.sml`		Provides an complete implementation of volatile database tables, including table creation, record creation, storage, deletion and modification, locking, indexes, etc. Works in virtually exactly the same manner as the sbme1 type but does not support the member operator. Written completely in SIMPOL. The speed is not blinding, but pefectly adequate when working with 1000 or so records with a few indexes per table. Performance should be tested for anything outside of these parameters.
`winfiledlg.sml`	X	Library for calling the open and save file common dialogs from another program, such as the older Superbase product. This will only work on a Windows NT-based operating system when called from a Win16 program such as classic Superbase. See the example program in the `samples\sbl` directory.
`xmllib.sml`	X	Provides a few useful functions when working with XML but not using the facilities of the Document Object Model that the `libxml.sml` library provides.

Part X. Programming Data-Aware Form Programs

This part discusses techniques for programming applications using the dataform1 family of types for implementing data-aware forms. It is expected that most applications of this nature will probably use the application framework library as their initial point of departure, but the knowledge from this chapter will work for any program that is working with the dataform1 type family.

Table of Contents

Chapter 22. Overview of Window and Dialog Types Provided with SIMPOL

This chapter will look at the various types and families of types supplied with SIMPOL. It will not go into excessive detail, but it will attempt to provide a clear view of the types, the hierarchy of types included by other types, and how each set of types was designed to be used. Types come in two varieties, those provided as C/C++ language components and those designed in SIMPOL itself. This section will also concentrate mainly on the GUI elements. For other parts, it may be useful to examine the source code to the libraries or check the Language Reference Guide. The types that will be discussed include both those from the C/C++-language based component WXWN, and also derived types built in the SIMPOL programming language. These include:

- wxwindow
- wxdialog
- dataform1
- printform1
- report1
- quickreport1
- graphicreport
- application
- appwindow

wxwindow

The wxwindow type is used to create the main window for an application (usually), and might contain a menu bar, tool bar, status bar and even child windows. The wxwindow type is also used to create child windows. Here are some of the other types that are directly associated with the wxwindow:

- wxmenubar
- wxtoolbar
- wxstatusbar

In addition to these types, there are also some functions that are important to working with top level windows:

- `wxprocess()`
- `wxbreak()`

A minimal program that presents a window with no content can be seen in the section called “Creating a Single Window”.

In order to respond to events, it is necessary to place the WX system into a state to respond to events. That is what the `wxprocess()` function does. It takes a time out value, which is typically set to `.inf`, the internal value for infinity. That means that unless it is forced to exit by some other method, the program will sit in that statement waiting for events forever. In the example, the task of exiting this state is fulfilled by the `quit()` function, which is called when the user clicks the close gadget for the window, selects Close from the system menu, or presses **Alt**+**F4** (in Windows). That results in a call to the `wxbreak()` function, the sole purpose of which is to terminate a `wxprocess()` function.

Note

The program could also call the `quit()` function for some other reason, such as a menu selection, a form button press, etc. that would result in the program exiting.

wxdialog

A dialog window is very similar to a main window, but with less features. It cannot have a tool bar, menu, or status bar. Also, dialog windows are in front of their parent window. The wxdialogtype in SIMPOL can be either modal or non-modal. Modal means that the dialog must be dealt with and dismissed before you can continue or click on the parent window. A non-modal dialog stays in front of the parent window, but the user can still click on the parent window. For examples of using wxdialog, see the section called “Working with Dialogs”.

wxform

In both the wxwindow and wxdialog types, the content is provided by the wxform type. The same form can be used in a window, a dialog, or even a toolbar (though the form should be sized and shaped appropriately). To place a form into a window or dialog, call the `setcontainer()` method of the wxform object passing the target window or dialog object. The form contains a ring of graphics and a ring of controls. Graphical elements are added to the form using the `addgraphic()` method of the wxform type. Controls are added using the `addcontrol()` method of the wxform type. The list of graphical elements supported includes:

- wxgraphicline
- wxgraphicrectangle
- wxgraphictriangle
- wxgraphicarc
- wxgraphicellipse

All of the above are type tagged as wxgraphic. This allows a variable that has been declared as `type(wxgraphic) g` to then contain a reference to any of the wxgraphic types. Graphical elements are *always* located behind controls. There is no method that can be used to cause them to be rendered in front of controls. The list of form controls currently provided is:

- wxformbitmap
- wxformbitmapbutton
- wxformbutton
- wxformcheckbox
- wxformcombo
- wxformedittext
- wxformgrid
- wxformlist
- wxformoption
- wxformscrollbar
- wxformsizebox
- wxformtext

All of these controls are type tagged as wxformcontrol, and therefore any variable declared as `type(wxformcontrol) c` can contain a reference to any of the form control types.

Note

The wxformoption type has a basic problem. It does not automatically come with any method of treating several of these buttons as a group. To overcome this, a solution was created and was placed in the `sbnglib.sml` library. This solution is based on the types:

- wxformoptiongroup
- wxformoptiongroupmember

To use it, create a wxformoptiongroup object. Then after creating each button, use the `ad-dmember()` method of the option group object to add it to the group. If you intend to assign an onchange event to the option button, do this first, since otherwise things won't work (when the option button is added to the group, its onchange event information is replace with that of the group, and the old information is stored so that it can be called later).

Iterating Through wxform Elements

Earlier it was said that the form controls and graphics are in a ring. A ring is a specific type of data structure. The supplied SIMPOL language library called `lists.sml` provides implementations of singly-linked lists and rings, and doubly-linked lists and rings, as well as a queue and a stack. It is also supplied in source code as the lists project in the `simpol\projects\libs` directory. The way this works is that a reference to the first control on the form is assigned to the wxform.firstcontrol property. The same is true of the first graphic. A reference to it is assigned to the wxform.firstgraphic property. Each control or graphic also has a property called next, which is a reference to the next graphic or control. The next property of the final control or graphic on the form will refer to the first one, thus creating the ring. If there is only one control or graphic on the form, then its next property will refer to itself. Below is a function that takes a wform type as a parameter and then returns an array of all the control names. It could just as easily use the same technique to change the colors of all the controls, or resize, them, etc.

Example 22.1. Iterating Through Form Controls

```
function getcontrolnames(wxform f)
  type(wxformcontrol) c
  array names
  integer i

  i = 0
  names =@ array.new()
  if f !@= .nul
    c =@ f.firstcontrol
    while c !@= .nul
      i = i + 1
      names[i] = c.name
      c =@ c.next
    end while c =@= f.firstcontrol
  end if
end function names
```

The same approach could be used for graphical elements.

When to Use wxform

Generally the wxform and its associated controls are a good choice for forms that will not have data directly associated with the controls. Utility programs are a good example, as are basic dialogs that just retrieve some user choices and then process the results.

dataform1

The dataform1 type was created in the SIMPOL language library called `databaseforms.sml` in order to provide a multi-paged, data-aware form system that works as a set of wrappers to the wxform types. One of the other enhancements is the support for system colors, so the page background color, plus the text and background colors of the controls also can take a system color identifier, which is interpreted at run time to decide which color to use. The dataform1 type family consists of the following types:

Table 22.1. dataform1

Type	Description
dataform1	This represents the entire form. The form contains a dring of dataform1page objects. Each page contains the graphics and controls that are found on that page. The form also has drings of controls, graphics, datasources, bitmaps, fonts, tables, links, and sibling links.
dataform1arc	A basic wrapper for the wxgraphicarc type, but also includes the necessary elements to be part of dataform1.
dataform1bitmap	A bitmap object that is also data-aware. Can be a static bitmap or it can have a control source that contains the path name of the bitmap, which can be a file system resource or located on the Internet using the HTTP protocol. Must be stored using the URL format: "`file:///c/mystuff/mypic`".
dataform1bitmapbutton	Provides a compatible wrapper for the wxformbitmapbutton type.
dataform1bitmapsource	Supplies the container for the bitmaps used in the form. Also contains the path and file name for images loaded from disk. Allows easy reuse of the same bitmap multiple times on one form (or potentially on multiple pages of a form).
dataform1button	Compatible wrapper for a button.
dataform1checkbox	Supplies a data-aware check box control that can not only have on or off, but which assign a value for the on state and another for the off state to the underlying database field.
dataform1combo	A very flexible data-aware combo box that can be filled with static values, read the values from an array, or retrieve them from a database table. Can also one set of values, but assign a different one. Finally, it also has the option to be filled by the user program by assigning a handler to the onfill event.
dataform1controlsource	The information used to connect a database field with a control, including the display format (used both to convert from dates, times, etc. to string for display in the control, as well as to convert from the string value to the target data type for storage in the field). Also contains information about a detail block, if the control is part of one.
dataform1datagrid	The purpose of the data grid is to make it easy to show data in a grid control from a table that is dependent on the main table in a form. It has restrictions on what can be done with it. The cells are read-only, since this grid is meant for display only.

Type	**Description**
dataform1datagridcolumn	This is a column in a data grid type. It contains the control source, which itself may contain a link to another table. An example might be a form with an ORDERS table, a data grid of information from the ORDERDTL table, and a grid column that is linked to the PRODUCT table and is showing the product name.
dataform1datasource	This type is used to hold the information necessary to reopen a data source, as well as a reference to the opened data source itself.
dataform1detailblock	A detail block is a collection of controls that are arranged in a single row, which is then replicated into a specified number of rows and columns and which may or may not have a scroll bar. This is a very powerful mechanism for displaying data. It can either be linked to the master table of a form, or it can be unlinked and filled using a SQL query.
dataform1edittext	The edit control is similar to the normal edit control, but has the control source and also has a speical feature for displaying a drop list based on an indexed search using a specified number of typed characters in the edit control. Works similarly to the effect seen in web browsers that remember previous form entries.
dataform1ellipse	A basic wrapper for the wxgraphicellipse type, but also includes the necessary elements to be part of dataform1.
dataform1grid	This is a free form grid control with which the programmer can create any functionality they wish. It is almost identical to the wxformgrid type but is compatible with dataform1.
dataform1line	A basic wrapper for the wxgraphicline type, but also includes the necessary elements to be part of dataform1.
dataform1link	This object contains the information necessary to create a join between two database tables. It is included in the dataform1controlsource type.
dataform1list	This is similar to the capabilities of the dataform1combo, allowing various ways to populate the list. Wraps the wxformlist type, but is limited to single selection, since the result will be assigned toa ssingle field in a database record.
dataform1option	A data-aware option button implementation. The interesting thing is that the control source is associated with the group, not the individual controls. A selection of an option will assign the value associated with the control to the field in the database record.
dataform1optiongroup	Provides the host for the control source for a group of option buttons. Also implements the functionality for grouping buttons together.
dataform1page	This contains the controls, graphics, and the reference to the wxform object that contains all the wxform controls and graphics that make up one page.
dataform1record	This is a container for a `type(db1record)` object that also contains a flag for indicating the record has been modified, plus methods for saving and deletion, plus events to call back to user program code when a record is saved and when it is deleted. These events are not typically used, however, instead the equivalent events of the dataform1 object are favored.
dataform1recordset	When working with a detail block, you may wish to retrieve the record(s) that make up a single row on the display. The dataform1detailblock method `getrowdata()` will return a record set with the records representing that

Type	Description
	specific row. A record set can also be used to update a row in a detail block, by calling the `setrowdata()` method.
dataform1rectangle	A basic wrapper for the wxgraphicrectangle type, but also includes the necessary elements to be part of dataform1.
dataform1scrollbar	This is a basic wrapper for the wxformscrollbar type to make it a proper part of dataform1.
dataform1table	This type contains a reference to a database table – type(db1table), it also contains an array of field information, a reference to the data source, a reference to the form, a reference to the current index, a dlistnode called parentnode which is used to traverse the tables on the form, and another dlistnode called datasourcenode that is used to traverse the tables in the data source. See below for information on iterating through elements on the form. There is a method called `gettablename()` that can be used to retrieve the table name of the table regardless of the type of table. It also has events for onnewrecord, onsaverecord, and ondeleterecord that can be defined by the application programmer to take an action at that point.
dataform1text	This type enhances the underlying wxformtext type by making it data-aware, so that information from a record can be shown, but not edited.
dataform1triangle	A basic wrapper for the wxgraphictriangle type, but also includes the necessary elements to be part of dataform1.

There are a number of design concepts that are associated with the dataform1 type. Among them are: every form has a master table and will typically have a master record. The master table is not meant to be changed once the form has been created and is in use. Normally only the master record (from the master table) can be modified. There are sibling links that connect to other tables in a 1:1 or n:1 relationship, like looking up a customer number in the customer table and then displaying the name on the form. There are also links to detail blocks and data grids, which are of the type 1:n, so if the current form contains an order, then the lines of the order might be in a data grid or detail block. When using the `deleterecord()` method, it only deletes the master record, it does not affect any linked records. For details on working with the dataform1 types see:Chapter 23, *Using Data-Aware Forms in SIMPOL*.

Using the Various dataform1 Services

The dataform1 type not only provides the capabilities of a form and its controls, but also adds database functionality to that. It includes methods for selecting records (when using linked tables, it also performs the lookups into those tables and refreshes the form automatically), changing pages, locking, editing, and saving records, firing off calculations and validations at the time records are saved, and provides two different approaches to managing data-entry. In this section we will discuss these various services. The interesting methods are listed below:

Table 22.2. dataform1 Methods

Method Name	Description
`blank()`	Clears all data from the data-aware controls on the form. Does not create a new record!
`check-dirtyrecords()`	Checks all associated records with the form and returns .true if there are any modified records.
`deleterecord()`	Deletes the current master record and then attempts to select the next record for display according to the current index. Must be locked first.

Method Name	**Description**
`discardrecord()`	Marks the current record as not modified (dirty) and unlocks the master record if it is locked. It will also call any programmer-defined ondiscard event handler.
`findcontrol()`	Given a string this function will search for a matching dataform1 control. If found, the return value will be a reference to the form control.
`findgraphic()`	Given a string this function will search for a matching dataform1 graphic. If found, the return value will be a reference to the graphic.
`getfieldandtable()`	This is passed field name and table name and if a matching table with a field of this name is found, it returns a reference to a dataform1controlsource object.
`lock()`	Call this method to lock the master record of the table. It also sets the modified (dirty) state to `.true`.
`nameinuse()`	To establish if a control name is in use (one name space is used for controls and graphics), call this function passing the name of the control. If in use it returns `.true`, otherwise it returns `.false`.
`newrecord()`	Creates a new master record for the form, also internally calls `blank()` to clear the form. It will call any previously defined onnewrecord event handler.
`refresh()`	This method re-reads the data from the records already selected and associated with the form and updates the data shown on the form.
`saverecord()`	This function should be called to save the newly created or modified master record. The record will automatically be unlocked unless the *`lock`* parameter is set to `.true`. Also, if an onsave event handler is defined, it must return `.true` if the save is meant to continue. Handling the onsave event is how the program can handle validations and calculations prior to the saving of the record. It receives the dataform1 object and an optional reference. To modify the fields of the target record, or retrieve values, use the `dataform1.masterrecord.record!fieldname` type of approach. If a validation fails, it is the responsibility of the programmer to either request a replacement value, or to set focus to a specific control and return `.false`, so that the record is not saved.
`selectcurrent()`	This works similarly to the standard `selectcurrent()` method of the ppcstype1record, though it works with sbme1record and vola1record as well. Also updates the form, any form links, and will call any defined onselect event handler (for calculated form content based on the data).
`selectfirst()`	This works similarly to the standard `selectfirst()` method of the ppcstype1record, though it works with sbme1record and vola1record as well. Also updates the form, any form links, and will call any defined onselect event handler (for calculated form content based on the data).
`selectkey()`	This works similarly to the standard `selectkey()` method of the ppcstype1record, though it works with sbme1record and vola1record as well. Also updates the form, any form links, and will call any defined onselect event handler (for calculated form content based on the data).
`selectlast()`	This works similarly to the standard `selectlast()` method of the ppcstype1record, though it works with sbme1record and vola1record as well. Also updates the form, any form links, and will call any defined onselect event handler (for calculated form content based on the data).

Method Name	Description
`selectnext()`	This works similarly to the standard `selectnext()` method of the ppcstype1record, though it works with sbme1record and vola1record as well. Also updates the form, any form links, and will call any defined onselect event handler (for calculated form content based on the data).
`selectprevious()`	This works similarly to the standard `selectprevious()` method of the ppcstype1record, though it works with sbme1record and vola1record as well. Also updates the form, any form links, and will call any defined onselect event handler (for calculated form content based on the data).
`setmasterrecord()`	Use this method to set a different record as the current master record of the form. This will then use the new master record to select any dependent records in detail blocks, data grids, and 1:1 links followed by a refresh().
`setmastertable()`	This will change the master table of a form. It should never be used in a normal application program. If you choose to use it, it should be used before creating any other controls, and the setmasterrecord() method should later be used to select a record unless the table is empty.
`showpage()`	To change pages on a mulit-page form, use this method.
`unlock()`	Call this method to unlock the master record of the form if it was previously locked.

There are more methods, many of which are associated with adding data sources, tables, graphics, controls, links, and so on, but which will not be discussed here. Since these are used by the code that loads the form, they are not as important as the ones used to actually work with the form once it has been loaded. If you are curious about the use of these, save a form as a dataform1 program and examine the source code, or look at the source code for the project `formlib.sml`.

printform1

The printform1 type was also created in the SIMPOL language library called `databaseforms.sml`. It provides a set of data-aware types for creating printable forms (forms primarily meant to be printed rather than displayed on the screen). This group of types shares a number of types with the dataform1 type group, and is type tagged as dataform1 as well as dataform1linkcontainer. For details about working with the printform1 family of types, visit Chapter 24, *Using Data-Aware Print Forms in SIMPOL*.

Table 22.3. printform1

Type	Description
printform1	This represents the entire printable form. The form contains a dring of printform1page objects. Each page contains the graphics and controls that are found on that page. The form also has drings of controls, graphics, datasources, bitmaps, fonts, tables, links, and sibling links.
printform1arc	A basic wrapper for the wxgraphicarc type, but also includes the necessary elements to be part of printform1.
printform1bitmap	A bitmap object that is also data-aware. Can be a static bitmap or it can have a control source that contains the path name of the bitmap, which can be a file system resource or located on the Internet using the HTTP protocol. Must be stored using the URL format: "`file:///c/mystuff/mypic`".
dataform1bitmapsource	Supplies the container for the bitmaps used in the form. Also contains the path and file name for images loaded from disk. Allows easy reuse of the same bitmap multiple times on one form (or potentially on multiple pages of a form).

Type	**Description**
dataform1controlsource	The information used to connect a database field with a control, including the display format (used both to convert from dates, times, etc. to string for display in the control, as well as to convert from the string value to the target data type for storage in the field). The detail block information is not used when it is part of a printform1control.
dataform1datasource	This type is used to hold the information necessary to reopen a data source, as well as a reference to the opened data source itself.
printform1ellipse	A basic wrapper for the wxgraphicellipse type, but also includes the necessary elements to be part of dataform1.
printform1line	A basic wrapper for the wxgraphicline type, but also includes the necessary elements to be part of dataform1.
dataform1link	This object contains the information necessary to create a join between two database tables. It is included in the dataform1controlsource type.
printform1page	This contains the controls, graphics, and the reference to the wxform object (if the form is being displayed) that contains all the wxform controls and graphics that make up one page.
dataform1record	This is a container for a `type(db1record)` object. The features for managing modification are not used in printform1, since it is not designed to allow user interaction.
printform1rectangle	A basic wrapper for the wxgraphicrectangle type, but also includes the necessary elements to be part of dataform1.
dataform1table	This type contains a reference to a database table – type(db1table), it also contains an array of field information, a reference to the data source, a reference to the form, a reference to the current index, a dlistnode called parentnode which is used to traverse the tables on the form, and another dlistnode called datasourcenode that is used to traverse the tables in the data source. See below for information on iterating through elements on the form. There is a method called `gettablename()` that can be used to retrieve the table name of the table regardless of the type of table. It also has events for onnewrecord, onsaverecord, and ondeleterecord that can be defined by the application programmer to take an action at that point.
printform1text	This type enhances the underlying wxformtext type by making it data-aware, so that information from a record can be shown, but not edited.
printform1triangle	A basic wrapper for the wxgraphictriangle type, but also includes the necessary elements to be part of dataform1.

report1

Another SIMPOL language type is report1, which is part of the library called `reportlib.sml`. It implements a set of functions and types to provide a basic reporting engine. The output of the report1 type is not specified, the output is handled by the calling program. The Quick Report and Graphic Report packages are both implemented by using the report1 type. The primary purpose in making the report1 type available is to allow the creation of custom report types by SIMPOL programmers. For details about working with the report1 package, visit the section called “Working with report1”.

Table 22.4. report1 Types

Type	Description
report1	The key element of the report system is this type. It is used to define and then run the report.
report1aggregate	This type is used for defining an aggregate calculation for a group or the entire report.
report1aggregatevalue	This type is passed to a function that is handling the aggregate. Unless you are implementing your own aggregate calculation, rather than using one of the ones supplied, you will not use this type.
report1group	Provides the capability of adding one or more groups to a report. Includes events for ongroupstart and ongroupend, to allow the event handler to process the output at the start of the group (group name, etc.) and again at the end (totals).
report1groupinst	This is the type that is passed to the event handling functions for ongroupstart and ongroupend.
report1inst	When a report is running, this type is created to contain the data about the currently running instance of the report definition. It is passed to the events of the report1 object.

The rest of the implementation is mainly the functions that have been pre-defined for handling the various aggregate types. These consist of:

Table 22.5. report1 Functions

Function	Description
report1_agg_getval_count()	This is the function that retrieves the value for the COUNT aggregate.
report1_agg_update_count()	This function is called to update the COUNT aggregate value.
report1_agg_getval_mean()	This is the function that retrieves the value for the MEAN aggregate.
report1_agg_update_mean()	This function is called to update the MEAN aggregate value.
report1_agg_getval_median()	This is the function that retrieves the value for the MEDIAN aggregate.
report1_agg_update_median()	This function is called to update the MEDIAN aggregate value.
report1_agg_getval_mode()	This is the function that retrieves the value for the MODE aggregate.
report1_agg_update_mode()	This function is called to update the MODE aggregate value.
report1_agg_getval_sum()	This is the function that retrieves the value for the SUM aggregate.
report1_agg_update_sum()	This function is called to update the SUM aggregate value.

The report1 type is quite powerful, but unless you want to implement your own special report mode, you may find the Quick Report and Graphic Report engines to be more suitable or easily used.

quickreport1

A much easier to use reporting type in SIMPOL is the quickreport1 type, which is found in the `quickreportlib.sml` library. This provides a wrapper around the report1 type that delivers output to window, printer, clipboard, HTML, CSV, and database (SBME format). Specific information about working with the quickreport1 package, can be found in the section called “Working with quickreport1”.

Table 22.6. quickreport1 Types

Type	Description
quickreport1	The key element of the quick report system is this type. It is used to define and then run the report.
quickreport1columninfo	For each column an element of this type is required to define the column characteristics, including: the starting horizontal position and the width, both in micrometers, the alignment, and whether the column content should wrap onto the next line.
quickreport1datasource	This is the return value from the call to add a data source to the Quick Report. The return value is passed to the code that adds a table, but is otherwise not generally used externally.
quickreport1table	The wrapper for the database table containing a link to the data source and thereby all information required to reopen the table at another time. This is primarily used internally, though it is the return value from adding a table.

The instance types from the report1 type are also used in the event handlers for the quickreport1. In addition they also receive the quickreport1 object. There are a number of functions associated with the running of a Quick Report, which are listed below:

Table 22.7. quickreport1 Functions

Function	Description
report1_quickreport_output_groupfooter()	This function is called at the end of a group to output any defined group information, typically the count and/or an aggregate value.
report1_quickreport_output_reportfooter()	This function is called at the end of the report to output any defined report information, typically the count and/or an aggregate value.
report1_quickreport_output_reportheader()	This function is called at the start of the report to output any report specific information. In practice this function is used to open output files or output file header information.
report1_quickreport_outputpageheader()	Outputs the defined header information, if any, at the start of each new output page. It is only called once for some output formats.
report1_quickreport_outputrow	This function is called once for each row of output. It also is responsible for determining if a group has ended, or the end of the page has been reached.
loadquickreport()	Loads a Quick Report from the XML storage format.
savequickreport()	Saves a Quick Report in the XML storage format.
convert_dpi_mcm()	Converts a measurement from pixels to micrometers at the current dot per inch value of the display.
convert_mcm_dpi()	Converts a measurement from micrometers to pixels at the current dot per inch value of the display.

In general, no use is made of the above-named functions by user programs, since the beauty of the Quick Report is its simplicity. Once the report has been set up, it just needs to be run. It also has various options such as displaying a progress gauge that can be enabled or not as desired. The simplicity is an advantage but also the only real drawback of this report. For much more freedom in the design of a report, it is necessary to use the Graphic Report.

graphicreport1

The most powerful report type in SIMPOL is encapsulated in the graphicreport1 family of types, which are found in the `graphicreportlib.sml` library. A fully banded report engine is contained in this set of types. The resulting output can be sent to either window or printer. The Graphic Report engine uses the printform1 functionality to create reports that can included multiple fonts, images, graphics, nested groups, and aggregate calculations. Specific information about working with the graphicreport1 package, can be found in the section called “Working with graphicreport1”. Graphic Reports implement a banded report writer. What this means is that each section of the report is treated as a band of information, an the various bands are put together to create pages. There are bands for page header, page footer, report header, report footer, and for each defined group, group header and group footer, and for the body of the report. If a band is not defined, then it will not impact the output. For each band an area can be defined that is represented using a graphicreport1formpage, and is populated by graphicreport1form controls and graphics.

Table 22.8. graphicreport1 Types

Type	Description
graphicreport1	The key element of the graphic report system is this type. It is used to define and then run the report.
graphicreport1arc	Represents an arc that can be placed on the band of the report.
graphicreport1ellipse	Represents an ellipse that can be placed on the band of the report.
graphicreport1form	The physical representation of the report output definition is contained in this form.
graphicreport1formbitmap	A bitmap element for a graphic report form.
graphicreport1formpage	Each band of the report is represented by one of these pages, and the controls are placed on the page.
graphicreport1formtext	This is a text control that can be placed on the band of the report.
graphicreport1formline	Represents a line that can be placed on the band of the report.
graphicreport1formrectangle	Represents a rectangle that can be placed on the band of the report.
graphicreport1formtriangle	Represents a triangle that can be placed on the band of the report.

The report1inst and report1groupinst types from the report1 type are also used in the event handlers for the graphicreport1 package and like with the Quick Report package, a graphicreport1 object is passed to the event handling functions. The list of exported functions can be found in the following table:

Table 22.9. graphicreport1 Functions

Function	Description
report1_graphicreport_output_groupfooter()	This function is called at the end of a group to output any defined group information, typically the count and/or an aggregate value.
report1_graphicreport_output_groupheader()	This function is called at the start of a group to output any defined group information, typically the name of the group and current `GROUP` value.
report1_graphicreport_output_reportfooter()	This function is called at the end of the report to output any defined report information, typically the count and/or an aggregate value.

Function	Description
report1_graphicreport_output_reportheader()	This function is called at the start of the report to output any report specific information. In practice this function is used to open output files or output file header information.
report1_graphicreport_outputpagefooter()	Outputs the defined footer information, if any, at the bottom of each new output page.
report1_graphicreport_outputpageheader()	Outputs the defined header information, if any, at the start of each new output page. It is only called once for some output formats.
report1_graphicreport_outputrow	This function is called once for each row of output. It also is responsible for determining if a group has ended, or the end of the page has been reached.
loadgraphicreport()	Loads a Quick Report from the XML storage format.
savegraphicreport()	Saves a Quick Report in the XML storage format.

As with the Quick Report system, external programs will not likely make any use of the event handling functions, since it is designed to "just work". Instead, a user program should define the onoutput event of the graphicreport1formpage object.

application

The basic approach to developing a program using the application framework is: initialize the program (create the application object and the first appwindow and display it to the user with whichever form, menu bar, status bar, and tool bar required), call the `run()` method of the application object and respond to events with event handling code, and finally cleanup when asked to exit.

The application framework provided with SIMPOL in the `appframework.sml` library file includes several data types and a number of functions that together with the items that are included in the `formlib.sml` library, such as: the dataform1 and printform1 families of types, plus the numerous functions included in the library for formatting types, working with databases, parsing information and other useful tasks, make it eay to create robust database-based applications. The key to this functionalty lies in the two main types: application and appwindow. Information that is universally required is associated with the application type. Information that is specific to a single window is associated with the appwindow type. Some useful information about working with the application framework can be found in Chapter 26, *Using the SIMPOL Application Framework*. Let's start by having a look at the application type:

Table 22.10. application Properties

Property	Description
localeinfoold SBLlocale	This contains the datelocale and the numlocale properties, that are used in formatting dates and numbers throughout the libraries.
dring datasources	This is a ring of data source objects using the datasourceinfo type. Data sources are stored at the application level since they need to be used by all aspects of the program. Single-user data sources like sbme1 cannot be opened multiple times even by the same program, so the intiial open is used by all elements within the program. Also, even though it is possible to open the same table more than once using PPCS, the objects would not be compatible even on the same table if used from more than one opened instance.

Property	Description
tdisplayformats displayformats	This contains a set of string properties. One for each of the default data types that might need conversion to or from string. The properties are: defboolean, defdate, defdatetime, definteger, defnumber, and deftime. They are designed to work together with the functions: `boolstr()`, `DATESTR()`, `datetimestr()`, `STR()` (used by both integer and number values), and `TIMESTR()`.
string inifilename	This property provides a placeholder for the name of the configuration file that might be associated with the program. See the `conflib.sml` for functions that work with configuration files.
localeinfo locale	This contains a more modern set of locale information, which can be of use in an application, though it is currently there for potential future expansion.
event onexitrequest	This event provides a mechanism whereby the application programmer can be called to determine if the application should close. If the user closes the last open window then in the `closewindow()` function if this event is defined, the user program will be called and the application will only be closed if the return value is equal to `.true`. Otherwise the window being closed will be redisplayed. The handler function takes the following parameter types: (*`application`*, *`wxwindow`*, *`type(*) reference`*). The final reference parameter will only be passed if it is defined.
integer ostype	These are currently defined in the application source file as: `OS_UNKNOWN 0`, `OS_WIN32 1`, `OS_LINUX 2`. Although every effort is made to ensure transparency between platforms, sometimes it is necessary to detect the platform.
ppcstype1 ppcs	This is the place holder for a ppcstype1 object that can be used throughout the application for opening PPCS-based tables. The system treats all tables coming from the same IP address and port as being part of a single data source.
boolean running	This property is set to `.true` when the system enters the `run()` method of the application object. The exit code in various places will set this property to .false so that it exits the run loop and exits the `run()` method.
sysinfo systeminfo	This type contains information about the environment and is initialized during the creation of the application object. This includes the size of the display, thickness of scroll bars, the list of system colors and their RGB values, and the system default font.
string title	This is the default caption for the windows of the application.
wxbitmap windowicon	To promote efficiency, this contains the bitmap used for the window icon in the various windows displayed by the program.
dring windows	This ring contains the appwindow objects that are created in the application. By using this ring all of the various application windows can be examined even if no appwindow is currently available. It also allows the programmer to iterate through all of the appwindow objects if they have one.

The basic approach to working with the application type is to create another application type specific to the user's program, such as myapplication, insert the application type into it as the first element marking it as `reference` and `resolve` (`reference` because it is important to call the `new()` method of the application type and `resolve` so that the properties of the application type appear to be built into the myapplication type). Also type tag the myapplication type as application. There are a number of functions that take an argument of *`type(application)`* and this is done so that a derived application type will still work when passed to the function.

appwindow

The most used type in the application framework is the appwindow. Unlike the application type, it is used as is, rather than having another type derived from it. This type is responsible for managing the database tables, opening forms, enabling and disabling menu and tool bar elements by using call backs via the onmanagemenu and onmanagetoolbar events and manages other window-specific information, such as the last selected unique key for the master table of the form, the current table, current directory path (for consistency when opening files and presenting file selection dialogs to the user), and provides a property that can hold a report so that it can be opened and retained within the window. Additional information about working with the application framework can be found in Chapter 26, *Using the SIMPOL Application Framework*. The details of the appwindow type are show below:

Table 22.11. appwindow Properties

Property	Description
type(application) app	Holds a reference to the application object (or a derived application object).
dlistnode appnode	This is the dlistnode node that is a member of the windows dring in the application type. It makes iteration through the appwindow objects possible.
string currentpath	The most recent path used to load or save a file (this starts at the current directory and is the responsibility of the application programmer to update).
tableinfo currenttable	The a tableinfo object containing the current table for the window. This is typically the same as the form's master table.
boolean disablewindowresize	Normally when the `openformdirect()` method is called, the window is resized to exactly fit the form. Set this to `.false` to disable this feature.
boolean fastselection	This should be set to `.true` if in the loop for a fast forward or rewind operation. Otherwise it is set to `.false`.
dataform1 form	A reference to the form that is currently loaded into the window.
string lastinternaluniquekey	This property is meant to be updated by the record selection code and assumes that the database tables have been opened such that the internal record ID is exposed (as is done in the appwindow.`opendatatable()` method. It contains the the most recent internal record key. This can be useful in various situations where a table does not have a unique index.
anyvalue lastselkeyvalue	This value is the most recent entry into the select key lookup dialog. It will be used to pre-set the prompt, so that the most recent entry reappears when searching, as long as the search index is the same. It should be cleared when changing indexes.
wxmenubar mb	This contains a reference to the menu bar for the window. It isn't necessary, since it can be reached via the wxwindow property w, but it is convenient.

Property	**Description**
event ondeleterecord	Use this event, together with the supplied function deleterecord() to provide any required special handling when deleting a record.
event onmanagemenu	If there are one or more entries in the menu that need to be enabled or disabled when any of a number of things occurs, such as adding, modifying, saving, or deleting a record, opening or closing a table or form, then by assigning a handler for this event, the code will be called. The function prototype is: (*appwindow appw*, *wxmenubar mb*, integer *eventtype*, string *tablename*, *type(*) reference*). The event types are listed below.
event onmanagetoolbar	If there are one or more entries in the tool bar that need to be enabled or disabled when any of a number of things occurs, such as adding, modifying, saving, or deleting a record, opening or closing a table or form, then by assigning a handler for this event, the code will be called. The function prototype is: (*appwindow appw*, *wxtoolbar tb*, integer *eventtype*, string *tablename*, *type(*) reference*). The event types are listed below.
type(*) report	This property provides a place holder for a report. It could be a Quick Report, a Graphic Report, or even one of your own derivations from report1. The advantage is that the report can be presented again to the user once defined.
wxstatusbar sb	Contains a reference to the status bar for the window. It isn't necessary, since it can be reached via the wxwindow property w, but it is convenient.
dring tables	This is the parent for the ring of database tables stored as tableinfo types.
wxtoolbar tb	This contains a reference to the tool bar for the window. It isn't necessary, since it can be reached via the wxwindow property w, but it is convenient.
type(wxcontainer) w	This property holds the reference to the actual wxwindow object displayed on the screen. It is declared as type(wxcontainer) so that it can hold a wxdialog as well as a wxwindow.

Here is the list of parameter values and the symbolic constant names used in the application framework source code that can be passed to the handler functions for the onmanagemenu and onmanagetoolbar events:

- (1) iEV_OPENFORM
- (2) iEV_CLOSEALL
- (3) iEV_CHANGECURRENTTABLE
- (4) iEV_NEWRECORD
- (5) iEV_SAVERECORD
- (6) iEV_CLOSETABLE
- (7) iEV_OPENTABLE
- (8) iEV_CHANGERECORD

Chapter 23. Using Data-Aware Forms in SIMPOL

This chapter will describe the general design of the dataform1 family of types, as well as some techniques for successful use of the set of types.

Note

This chapter will not make much sense unless you have already read and feel comfortable with the earlier chapters covering variables and grammar.

The Design of dataform1

The goal of the design of dataform1 was to create a set of wrapper types for the wxWidgets-based form controls in order to provide a multi-page, data-aware form system similar in scope to that provided by Superbase. To do this required managing quite a bit more information than is need to just provide the wxform family of types. To work effectively, the data-aware forms would need to keep track of the data sources used, the database tables used, the current master record for the form, and for efficiency of implementation also the bitmaps used (there is no point in loading a large logo bitmap used on every page once for each page, it is more efficient to load it once and use it on the various pages). Also, the data-aware form would be a container of pages, where each would contain a wxform object to actually host the controls. Also, each control would need to be enhanced to allow it to store the information necessary to connect it to a field in the database. To finish it off, the functions would need to exist to carry out the required actions: selecting records, reading data from the record and updating the various controls, reading data from the controls as they are updated by the user and writing that back to the record, locking in both single and multi-user modes, and numerous things that might not be obvious in the first instance.

There are several different kinds of data types used in the dataform1 family. One is the wrappers of the controls, another the wrappers for the graphics, and finally additional internal and exported types used for the actual implementation of the public interface. Let's have a look at them here, starting with the graphics.

Graphical Elements

The graphical elements in the dataform1 family are fairly thin wrappers around the wxformgraphic controls. What they add are the necessary components to link them together and additional information, such as the support for named colors (colors that are not fixed but that are based on the operating system settings for that user, and therefore use symbolic names such as "Button Face" or "Window Text".

- dataform1graphicline
- dataform1graphicrectangle
- dataform1graphictriangle
- dataform1graphicarc
- dataform1graphicellipse

All of the data types named above have a common type tag, called `dataform1graphic`. All the graphical elements are located in a dring property (doubly-linked ring) called graphics. Both the dataform1 and the dataform1page have a property with this name. The form contains a ring of all graphics from all pages. The page ring contains the graphics from that page.

Form Controls

Unlike the graphical elements, the form controls often contain quite a few more properties and methods than the original wxformcontrol objects. This is because of the requirement to store information about their binding to a database field and table, the required display format, whether the control is part of a detail block (a set of controls displayed in rows where each row represents a record in a related database table), and if so in which row it is. Some controls require even more information. The dataform1list and dataform1combo controls allow quite a bit of flexibility in determining where the data comes from (including showing one value in the list but assigning another to the control). The dataform1datagrid type provides a grid that has columns associated with fields from a table, and it can be linked like a detail block to the master table of the form. Here is the list of form controls:

- dataform1bitmap
- dataform1bitmapbutton
- dataform1button
- dataform1combo
- dataform1checkbox
- dataform1datagrid
- dataform1edittext
- dataform1grid
- dataform1option
- dataform1list
- dataform1scrollbar
- dataform1text

The form control types are all type tagged `dataform1control`. They are located in a dring property (doubly-linked ring) called controls. Both the dataform1 and the dataform1page have a controls property. The form contains a ring of all controls from all pages. The page ring contains the controls from that page.

Utility Types

There are a number of important utility types that play various roles in making the whole package work. Here is a list of them:

- dataform1bitmapsource
- dataform1controlsource
- dataform1datagridcolumn
- dataform1datasource
- dataform1detailblock
- dataform1link
- dataform1optiongroup
- dataform1page
- dataform1record
- dataform1table

The dataform1bitmapsource type was created to store the original location of a bitmap that is loaded into a form. The reason for it is simple, without that information it would be impossible to save the form later and to know what value to store in the output file for the location of the image. For each image used on a form, a bitmap source object is created and is then associated with the resulting bitmap so that it can be found later.

The same is true of the dataform1controlsource type. This stores the actual field reference associated with the control, plus the dataform1table object, and optionally a display format. This information is necessary in order to read from and write to the database field and to correctly display and interpret the data in the control itself.

A dataform1datagridcolumn is similar in that it also stores information about its control source, it may also store a reference to a dataform1link object if the column is not from the master table of the grid.

The dataform1datasource stores information about the data source, either its file name and path, or its IP address and port number. It also contains a list of the database tables that are part of the data source.

One of the most complex objects is the dataform1detailblock, which is a special type of container that provides a replicated group of controls, in rows and columns, that can be linked to the master table of a form. It can work in two different ways: either as a block of rows of data (records) that are related to the master record of the form, or else as a completely independent block of data, the content of which is governed by a query. In both cases, the data is read only (from the user's perspective). There are features in the design that allow for retrieving the database record for a given row, for updating that record or even replacing it, and also for removing the row from the result set. It also contains methods for scrolling the block up or down, a page or a row at a time. The detail block is currently not optimized for reducing the records read, so if the link results in reading 100,000 records, then it will do so, delaying everything until it is done. As such, it is important to choose the links and data design wisely.

The dataform1link contains the information that links two tables together, and it also stores the record sets that are read as a result of using the link to read records.

The dataform1optiongroup acts as the management object for a group of dataform1option controls. In this special case, the data source is associated with the group object, and not with the controls. It also ensures that if one option button in the group is selected, that the others are deselected.

The dataform1record contains properties that assist it in knowing if the record has been changed, but not yet saved, and provides a place to define events such as onsaverecord or ondeleterecord.

Similarly, the dataform1table object stores information about the current state of the table, such as the current index, an array of field information including display formats, and events like onnewrecord, onsaverecord, and ondeleterecord.

The dataform1page is the container of all items specific to a single page of the form.

Iterating Through dataform1 Elements

The technique for iterating through dataform1 elements is slightly different to that used in the wxform. Firstly, there are many different dring properties: controls, graphics, bitmaps, tables, datasources, detailblocks, links, siblinglinks, obgroups, and pages. Iterating through these drings is fairly consistent, but you need to know what to expect from each one, so that the varialbe used to hold the current item is correctly defined. Below is a table showing the dring and the type that a variable must be declared as in order to hold any given member of the dring.

Table 23.1. dataform1 dring Types

Ring Property Name	Required Type
controls	type(dataform1control)
graphics	type(dataform1graphic)
bitmaps	dataform1bitmapsource
tables	dataform1table
datasources	dataform1datasource
detailblocks	dataform1detailblock
links	dataform1link

Ring Property Name	Required Type
siblinglinks	dataform1link
obgroups	dataform1optiongroup
pages	dataform1page

In each case the approach is the same:

Example 23.1. Iterating Through dataform1 dring Properties

```
function collectdf1controlnames(dataform1 f)
  type(dataform1control) c
  string names

  names = ""
  c =@ f.controls.getfirst()
  while c !@= .nul
    names = names + c.name + "{d}{a}"
    c =@ c.formnode.getnext()
  end while c =@= f.controls.getfirst()
end function names
```

In each case, the code tends to look very similar. It starts by getting the first item in the ring, then if that is not null (the ring has at least one entry), it enters the loop, processes whatever it is doing (the purpose for going through all the entries), then retrieves the next one in the ring, finishing when it has reached the first one again. In the prior example, since any number of different control types will be returned by the call to `c.formnode.getnext()`, the variable `c` is declared with the method used for defining a variable that can contain a type-tagged group of types. The type tag dataform1control is not a type, it is a type tag associated with each dataform1control in its type definition, to enable exactly this sort of functionality. For further information about type tags see the section called “Value Types, Reference Types, and Type Tags”.

Most of the types have a formnode property, which contains the reference to the dlistnode that makes the item part of the ring. Some items have a different name, and some have more than one node, so selecting the correct one is essential. For example, the form controls have a formnode and a pagenode (the dataform1option control also has a groupnode). To iterate through all the controls in the form, start with the dataform1 controls dring and use the formnode of each control to get the next one. To iterate through all the controls on a given page, use the controls property of the dataform1page and then use the pagenode of each control to get the next one. Here is an example that iterates through all pages on a form, and through each control on the page.

Example 23.2. Iterating Through the Controls on Each Page of a dataform1

```
function df1pagesandcontrols(dataform1 f)
  dataform1page p
  type(dataform1control) c
  string info

  info = ""
  p =@ f.pages.getfirst()
  while p !@= .nul
```

```
    if info > ""
      info = info + "{d}{a}"
    end if
    info = info + p.name + "{d}{a}" + "-" * .len(p.name) + "{d}{a}"
    c =@ p.controls.getfirst()
    while c !@= .nul
      info = info + "  " + c.name + ": type=" + c.type + "{d}{a}"
      c =@ c.formnode.getnext()
    end while c =@= p.controls.getfirst()
    p =@ p.formnode.getnext()
  end while p =@= f.pages.getfirst()
end function info
```

In the preceding program the two iteration variables are `p` and `c`. The page variable is defined to be of one specific type: dataform1page, since that is the only type that is managed by the ring. The other is defined as type(dataform1control), since all of the various dataform1 control types will be in the ring and therefore the variable needs to be able to hold a reference to any of them.

Controlling with Events

It will usually be as a result of your program code calling the dataform1 methods: `selectfirst()`, `selectlast()`, `selectnext()`, `selectprevious()`, etc., that will result in data changing on the form. The onselect event can be assigned a handler so that you can run some code each time a record is selected. This can be used to implement calculated form content based on the value of the underlying record. The onsave event can be handled to implement validations and calculations before the record is saved, or to refuse to save if the validation fails. In the same way, the onnewrecord event can be used to implement default values for the new record, and the ondiscard event can be used to do cleanup if the user chooses not to save a record.

Using the Special Features

There are a number of special features that can provide more user-friendly and powerful applications and which are included in the design of the dataform1 family. In this section we will discuss these features.

The `onfill` Event

The list types, dataform1combo and dataform1list both include a special event called `onfill`. What this event does is that at the point where the code would normally fill the list of the combo or list box, if this event has a handler assigned, it will instead call that handler. This gives the programmer the ability to fill the list themselves, potentially using content that would otherwise be difficult to define in the normal approach.

Note

It is important to note here that the `onfill` event is normally only called when the form is loaded.

The Drop List For Edit Controls

The dataform1edittext control has a special feature that may be familiar to some from the technique in web browsers. Namely, in the edit control while typing suddenly a list will drop down containing related content that had been typed into the box in the past. As the user types, it filters the content such that the beginning of each list entry matches the content that has been typed into the box. The user can then select an item from the list using the mouse, or in our case tab into the control and use the arrow keys to go up and down the list. As they change entries in the list, the text in the edit control is updated to match the entry. When they tab to the next control, the list vanishes.

This sort of functionality is available to every dataform1edittext control. The list content *must* be retrieved from a database table. This feature cannot be stored in the form definition when it is saved, it must be added after the form has been loaded. To use it, call the `enabledroplist()` method. This method takes the following parameters:

1. boolean ***`enable`***
2. type(db1index) ***`index`***
3. integer ***`activationcharcount`***
4. integer ***`listheight`***
5. integer ***`maxsearchentries`***
6. integer ***`error`***

To turn on the functionality, the method must be called with the ***`enable`*** parameter set to `.true` and the ***`index`*** parameter must be a valid index object for the index of the table on which to search. The remaining parameters are optional and have usable default values. The ***`error`*** parameter should always be supplied and tested before making use of the functionality. The names of the parameters should make clear what they do, but here is a brief description anyway:

- ***`activationcharcount`*** – This setting determines how many characters must be typed into the control before the search functionality is activated. If you have a large table, it may be worth sticking to the default of 2 characters or even increasing it slightly.
- ***`listheight`*** – This value determines how tall the list box will be. It will be located directly below the edit control and will be the same width as the edit control. The list cannot extend beyond the form height and any value that causes this will be automatically adjusted.
- ***`maxsearchentries`*** – This is a very useful setting that allows the programmer to limit the number of successfully found matching entries. This should be set to some useful value in the range 20-100 probably. It prevents lag while the searching is carried out. Since the user can simply type another character to search again with a finer filter it is no burden to keep the list size low.

Using a Query to Fill a Detail Block

The dataform1detailblock normally must be linked to the master table of the form. There is a new mode that has been added to allow an unlinked detail block to be managed by the programmer. The detail block can be filled using a query. This can be very useful to see the current status of a selection of the data. For example: unfulfilled jobs, uncleared bookings, completed jobs, open orders, etc. These could then be further filtered to show only those from today, the last hour, the past week, etc.

To use the data grid in this way, call the `setparams()` method and assign the ***`usequery`*** parameter to `.true` and the ***`whereclause`*** to the WHERE clause that you wish to be applied to the master table of the detail block. The call the `runquery()` method, passing in the error parameters (so that you can see if your query was valid or not). Once the query has been run, the result set will be stored and the first page of the detail block will be filled. The SQL92 syntax information can be found in the section called “Using SQL92 in SIMPOL”.

Two Approaches to Working with dataform1

The basic approach to working with data-aware forms in systems like SIMPOL or Superbase, is to lock a record prior to modification, allow the user to edit the record, and then save the record (unlocking it in the process) or to unlock the record (as a result of selecting another record). Systems that do not implement record locking (or that can not use it efficiently) such as most SQL database systems, take a different approach. They allow the user to make changes to a record, and then only when the time comes to commit the changes, they lock the record, check to see if it has changed since it was originally read, and if not, they commit the changes and unlock the record. The problems begin to arise if the record has changed in the interim.

In this section, we will stick to the former style of working, but even that has two different approaches. One is to use the auto-locking approach (the default in dataform1), the other is to use the explicit locking approach. Sophisticated systems (and especially multi-user systems) are likely to require the explicit approach. Let's have a look at each one in more detail.

Auto-locking

This is the easiest approach, since it just works. If the user clicks on a control that is not read-only or disabled, the control receives focus. If the user changes the content, when the focus leaves the control, the dataform1 system will attempt to lock the record. When the user saves, then everything is automatically committed. If the user goes elsewhere, the changes to the record will be discarded, it is up to the programmer to check to see if the record needs saving. This is integrated into the `appframework.sml` functionality, see the later section for more information. It is important to note, because of a curious issue with the wxWidgets library, the SIMPOL wxform.`clearfocus()` method has a quirk. Even though it clears focus from the form, wxWidgets stores the control that previously had focus. If the user then tabs away and tabs back again, the previous control to have had focus will have it again. We are looking into this with a view to changing that behavior, but until then it is important to consider what impact that may have on your application.

Auto-locking

An alternative approach, necessary to anyone who intends to provide user-level access control methods, is explicit locking. To use this, the dataform1 object has auto-locking turned off, and the *prevent focus* functionality enabled. This means that when the user attempts to place focus on the form, it fails, although buttons can still be pushed. The correct approach might look like this, where the variable `f` contains a dataform1 object:

```
// after opening the form using appwindow.openformdirect()
f.autolocking = .false
f.preventfocusmode = .true
f.preventfocus = .true
```

Once this has been set up, the `appframework.sml` library and the dataform1 package will handle the rest. When you wish to allow editing, for example via a menu or tool bar event, calling the `modifyrecord()` function will allow the user to change the data on the form. Once the form is saved, the dataform1 object will once again prevent focus on the form.

Making Use of `formlib.sml`

The `formlib.sml` library contains a copy of the `databaseforms.sml` library, and so if you add `formlib.sml` to your project, you don't need to add `databaseforms.sml`. There are quite a few types and functions in the library, but only a few are of any real relevance. The two most likely to be used functions are: `opendataform1()` and `savedataform1()`. The first opens a dataform1 from a file, and the second saves an existing dataform1 as a file. SIMPOL forms are stored in XML format, which is a standard text file that can edited in any editor, such as `notepad.exe`. When opening a dataform1 using the `opendataform1()` function, there are a large number of parameters that can (and should) be supplied. These include the defaults for various display formats, a dring of data sources that may be already open, and an array of database tables that are already open. Any database table that is passed in will prevent a database table of the same name being opened using the data source information stored with the form. This approach means that the form can be created using the single-user engine but can be opened using a previously opened set of tables that are being accessed using the multi-user engine.

The data sources are expected to be in a dring of datasourceinfo objects (these are not dataform1datasource types, but they are similar). The array of tables is expected to begin at 1 (like all arrays in SIMPOL) and to consist of entries of type(db1table). Both of these are easily retrieved when using the `appframework.sml` library and architecture. See the next section for details.

Chapter 24. Using Data-Aware Print Forms in SIMPOL

The printform1 family of types provides the ability to design, save, load, and print data-aware forms with an accuracy to the nearest micrometer. This chapter discusses how to use the set of types that implement this functionality.

Note

As is the case with other chapters, this chapter will not make much sense unless you have already read and feel comfortable with the earlier chapters covering variables and grammar.

The Design of printform1

The approach to printform1 was to provide a method of printing accurate forms to the print preview window and to the printer, without needing to previously display the printed form to the user. Although it is possible to display the printed forms, all coordinates are stored in micrometers and are then converted as well as possible to pixels for display purposes. At the time of writing, no significant testing has been done with the display of these forms. That has been reserved for the period of time when the Print Form Designer is being developed.

The first step in working with the printform1 family of types is to learn the members of the family and what role they play. Here is a list of the types:

- printform1
- printform1page
- printform1graphic
- printform1control
- printform1arc
- printform1ellipse
- printform1line
- printform1rectangle
- printform1triangle
- printform1text
- printform1bitmap

Understanding the list is fairly easy. The first element is the form, the second represents a page on the form. The third is a generic type that incorporates most of the elements that the graphic controls have in common, the same is true of the fourth item, but for controls. This was done because it turns out that the SIMPOL IDE is able to provide context help for variables that are declared using a type tag if that type tag name is also defined as a type. All of the graphic elements: the arc, ellipse, line, rectangle, and triangle, are type tagged using printform1graphic, and the text and bitmap items with printform1control, which greatly eases the development in the IDE of applications that use variables based on the type tag.

Each of the graphic types incorporate the related wxgraphic type, for use when displayed. In addition, they have a duplicate of the all the properties that affect the final look, prefaced in most cases with the word "`print`". The reason for this is that the design required that each of the elements carry the most accurate units, plus that it should be possible to have the entire form created without needing access to the embedded wxWidgets control, since that would require the control to be created in some displayable form. Since it was necessary to be able to draw the control directly into a wxprintout object, all of the elements that affected that result needed to be part of the property list for the graphic or control.

Let's go through each of the properties of the types, to see how they are constructed. The first and most complex of these is the printform1 type. It has many similarities with the dataform1 type. So many, in fact, that the original dataform1 type was also type tagged with dataform1, and the printform1 also carries this type tag, as well as the dataform1linkcontainer tag. Doing this allowed printform1 to reuse many of the types used in the dataform1 family of types, such as: dataform1datasource and dataform1link. Here is the type definition:

```
type printform1 (printform1, dataform1, dataform1linkcontainer) \
                export
  embed
  printform1private _private
  boolean valid
  integer defpagewidth
  integer defpageheight
  integer defpagebackcolor
  boolean designmode
  boolean dirty
  boolean createdisplayform
  boolean locked

  SBLNumSettings defnumericlocale reference
  SBLlocaledateinfo defdatelocale reference
  string defnumberformat
  string defdateformat
  string deftimeformat
  string defdatetimeformat
  string defintegerformat
  string defbooleanformat
  string name
  string filename
  string printpreviewtitle

  dring datasources
  dring tables
  dring bitmaps
  dring controls
  dring graphics
  dring pages
  dring siblinglinks

  event onselect
  event onsave
  reference
  type(*) _
  type(*) __ resolve
  dataform1table mastertable
  dataform1record masterrecord
  type(wxcontainer) container
  wxfont deffont
  printform1page currentpage
  array fonts

  function addbitmap
  function addcontrol
  function addgraphic
  function adddatasource
  function addpage
  function addtable
  function blank
  function builddisplayform
  function clearsiblinglinks
  function findbitmapsource
  function findcontrol
  function finddatasource
  function findgraphic
  function findsiblinglink
  function findtable
  function getfieldandtable
  function getfont
  function lock
  function nameinuse
  function print
  function refresh
  function removedisplayform
  function saverecord
  function selectcurrent
  function selectfirst
  function selectkey
  function selectlast
  function selectnext
  function selectprevious
  function setcontainer
  function setdirtystate
  function setmastertable
  function showpage
  function unlock
end type
```

Much of the type definition of printform1 is taken from that of dataform1, so let's look only at the differences. The createdisplayform indicates if, while creating the form, it should also create a displayable form using wxform and related types. There is also a printpreviewtitle property, which is used as the caption of the print preview window if the form is printed to that destination. Although there is an onsave event, this type was not designed for doing data-entry, and it may be removed at a later date. Like the dataform1 type, it contains numerous default properties and various rings of controls, graphics, tables, data sources, and the like. The methods are also quite consistent with those used in dataform1. One significant difference is the `builddisplayform()` method. This can be called after the form has been created to produce the display version of the form, by converting the print coordinates to display versions. The other method that is new is `print()`. This method takes a boolean parameter called *`showprintpreview`* that defaults to `.true`. It also takes a *`dialogdata`* parameter that can contain the printer information, so that the print dialog does not need to be shown to the user. This can be very handy for unattended printing.

The next type we wish to look at is the page. Like with the form, the page is very similar to the dataform1page type and it has also been type tagged as dataform1page. Here is the type definition:

```
type printform1page(dataform1page) export
  reference
  wxform wxformpage

  embed
  dring controls
  dring graphics
  integer pagenum
  string name

  integer printbackgroundrgb

  // These are the actual values for the printout in micrometers
  integer printwidth
  integer printheight

  reference
  type(*) _
  type(*) __ resolve
  dlistnode formnode
  printform1 form

  function addcontrol
  function addgraphic
  function builddisplayform
  function changename
  function print
  function resize
  function setactive
end type
```

The significant differences are again specific to the print capabilities. The printbackgroundrgb property contains the background color. The printwidth and printheight properties contain the paper size that will be passed in when producing the wxprintpagetemplate. As with the form, there are also the two methods: `builddisplayform()` and `print()`. The first is called to create the display version of a single page. The second is called to transfer the page to a printout, which is passed to it by the form version of `print()`.

Adding graphics and controls to the print form is very much the same as adding them to a dataform1 object. To add a graphic call the `addgraphic()` method, and to add a control call the `addcontrol()` method. The parameters to each are slightly different to those for the dataform1 versions, and are worth a look. Here is the parameter list for the printform1.`addgraphic()` method:

1. type *`graphictype`*

2. point *`printpoint1`*
3. point *`printpoint2`*
4. point *`printpoint3`*
5. point *`printmidpoint`*
6. integer *`rgb`*
7. integer *`borderrgb`*
8. integer *`width`*
9. integer *`borderwidth`*
10. boolean *`visible`*
11. boolean *`bordervisible`*
12. string *`printname`*
13. type(printform1graphic) *`next`*
14. printform1page *`page`*
15. integer *`error`*

As we can see from the preceding description, most of the parameters have the same name, but the point parameters differ. This was partly because in the original design it was possible to pass both the display and the print parameters in, until it was redesigned to always calculate the display parameters from the print ones. The same thing is true in most ways with the printform1.`addcontrol()` method. Here are the parameters for that:

1. type *`controltype`*
2. integer *`printleft`*
3. integer *`printtop`*
4. integer *`printwidth`*
5. integer *`printheight`*
6. string *`text`*
7. boolean *`visible`*
8. wxbitmap *`bitmap`*
9. string *`scaling`*
10. integer *`backgroundrgb`*
11. integer *`textrgb`*
12. string *`printalignment`*
13. wxfont *`font`*

14.string *`printname`*

15.boolean *`backgroundvisible`*

16.boolean *`undergraphics`*

17.boolean *`underbitmaps`*

18.type(printform1control) *`next`*

19.printform1page *`page`*

20.type(db1field) *`field`*

21.dataform1table *`table`*

22.string *`displayformat`*

23.integer *`error`*

Again the majority of the parameters are the same as those from the dataform1.`addcontrol()` method, but a few differences are clearly visible. All of the position parameters have the word "`print`" as the first part of their name, as do the alignment and name parameters. For the alignment that is because the type of alignment control allowed on a wxprintout is more intricate than that on a wxform. Some of the other new parameters are specific to capabilities of the wxprintbitmapitem and wxprinttextitem types. Rather than go into each of the graphics and controls, since they are very similar to the standard ones, it is probably better to just look at how to create and print a form. The next section will do just that.

Working With printform1

In this section we will create a small program that demonstrates using each of the controls on a printed form. Learning from an actual program is generally the best approach. The following sample creates a print form, populates it with controls and graphics, and then prints it to the print preview window. It does not use any of the data-aware features of the controls, but doing so is trivial, it merely requires also creating and adding the data sources and tables, and then assigning fields and display formats to the controls. That is identical to the way it works in normal data-aware forms, so for the purpose of this demonstration, it will be left out. Here is the sample code:

```
function main()
  printform1 pf
  printform1page page
  type(printform1graphic) g
  type(printform1control) c
  integer e
  wxwindow w
  wxbitmap bmp
  wxfont font
  string s, url

  e = 0
  w =@ wxwindow.new(1, 1, 300, 200, \
                  captiontext="Close me when done", error=e)
  if w !@= .nul
    w.onvisibilitychange.function =@ quit
    pf =@ printform1.new(error=e)
    page =@ pf.addpage(210000, 297000, 0xffffff, name="ptest", \
                       error=e)
    g =@ pf.addgraphic(printform1line, point.new(30000, 30000), \
                       point.new(180000, 30000), width=100, \
                       rgb=0, printname="l1", page=page, error=e)
    g =@ pf.addgraphic(printform1rectangle, \
                       point.new(30000, 50000), \
                       point.new(100000, 70000), borderwidth=100,\
                       rgb=0xff00, borderrgb=0x0, printname="r1",\
                       page=page, error=e)
    g =@ pf.addgraphic(printform1triangle, \
                       point.new(110000, 50000), \
                       point.new(140000, 50000), \
                       point.new(125000, 80000), borderwidth=100,\
                       rgb=0xff, borderrgb=0x0, printname="t1",\
                       page=page, error=e)
    g =@ pf.addgraphic(printform1arc, point.new(86519, 206056), \
                       point.new(179917, 206321), \
                       printmidpoint=point.new(133350, 154198), \
                       borderwidth=100, rgb=0xff0000, \
                       borderrgb=0x0, printname="arc1", page=page, \
                       error=e)
    g =@ pf.addgraphic(printform1ellipse, \
                       point.new(68281, 229369), \
                       point.new(115113, 249213), \
                       printmidpoint=point.new(68281, 249213), \
                       borderwidth=100, rgb=0xff00ff, \
                       borderrgb=0x0, printname="ellipse1", page=page,\
                       error=e)
    // The image is 192x80
    url = "http://www.simpol.com/images/style1/logo.png"
    bmp =@ retrievebitmap(url, "png", error=e)
    if bmp =@= .nul
      bmp =@ createblankbmp(192, 80, missing=.true, error=e)
    end if

    if bmp !@= .nul
      c =@ pf.addcontrol(printform1bitmap, 45000, 70000, 50800, \
                         21167, bitmap=bmp, \
                         scaling="preserveaspect", page=page, \
                         error=e)
    end if

    s = "The quick brown fox jumped over the lazy dog. \
         Peter Piper picked a peck of pickled peppers. How \
         many pickled peppers did Peter Piper pick?"
    font =@ wxfont.new("Arial", 13, "n", "n", "", error=e)
    c =@ pf.addcontrol(printform1text, 40000, 130000, 90000, \
                       35000, s, backgroundrgb=0x0, \
                       textrgb=0xffffff, printalignment="", \
                       font=font, printname="text123", \
                       page=page, error=e)
    pf.print(error=e)
    wxprocess(.inf)
  end if
end function


function quit(wxwindow me)
  wxbreak()
end function
```

The preceding source code creates a window (just to keep the print preview window open and the program running) and then create the print form. It adds one of each of the graphic types to it (please note that coming up with valid coordinates for an arc or ellipse is not trivial — these were converted from pixel values after drawing them using the SIMPOL Form Designer in SIMPOL Personal). Following the graphics, an image is retrieved from the Internet via the URL using the `retrievebitmap()` function that is part of the `databaseforms.sml` library. If it fails to retrieve a bitmap, it calls another function from the library to create a blank image with an X through it in the same size, to act as a missing image replacement. It then adds the bitmap to the print form. Finally it adds a text element with static text that is centered both horizontally and vertically and display as white text on a black background. This is then sent to the print preview window. Once the small main window is closed, the program ends.

printform1 Summary

During this chapter we have discussed the purpose and design of the printform1 family of types. We have also learned how they are similar and how they differ from the dataform1 family of types. Using a small sample program we have seen how to create and print a form using program code, as well as how to retrieve a bitmap using a URL from a web server on the Internet or in an Intranet.

Chapter 25. Using Reports in SIMPOL

In this chapter the four report engines will be discussed: `sql1.sml`, `reportlib.sml`, `graphicreportlib.sml`, and `quickreportlib.sml`. The second makes use of the first one, and both the Quick Report and Graphic Report engines make use of the the report engine, so much of what will be written about that engine applies to all of them.

Note

This chapter only discusses using the report engines programmatically.

Using the sqlq1 Type Directly

The sqlq1 type is where the true work for all the report engines takes place. This is the SQL92 engine. This engine supports a subset of SQL92 that is related to retrieving data from the database. It has a selectclause and a whereclause property. To run a query, there must be at least one column named in the selectclause. Then, the `prepare()` method must be called and assuming it did not generate an error, the results can be retrieved by calling the `getrow()` method until it returns `.false`. When using the various report types, you do not normally call the `getrow()` method, instead you would call the `run()` method. This will carry out the report, which will call the various event handling functions to produce the desired result.

The sqlq1 type is not normally used directly, though it can be quite handy. The `drilldown()` function from the `drilldown.sml` as well as the filter functionality for unlinked dataform1detailblock types both use this type.

Using SQL92 in SIMPOL

The SQL92 syntax supported in the sqlq1 type is:

- [TABLE_NAME.]COLUMN_NAME, [TABLE_NAME.]COLUMN_NAME AS
- AND OR
- = > < >= <= <>
- [NOT] LIKE '' ESCAPE
- unary +, unary -, +, -, *, /, || (string concatenation)
- POSITION(<string> IN <string>)
- EXTRACT(YEAR | MONTH | DAY | HOUR | MINUTE | SECOND FROM <date-time-or-datetime>)
- CHAR[ACTER]_LENGTH(<string>)
- UPPER(<string>)
- LOWER(<string>)
- SUBSTRING(<string> FROM <start-position> [FOR <length>])
- TRIM([[LEADING | TRAILING | BOTH] [<trim-char>] FROM] <string-to-trim>)

- CAST(<value-expression> AS <data-type>)
- ABS(<numeric-expression>)
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP

The COLUMN_NAME can be surrounded by double quote characters ("). This can be useful if the field name in the table contains one or more spaces (not recommended).

Here are some additional notes about working with dates, times, and datetimes:

- Dates must be supplied in the format yyyy-mm-dd when expecting to evaluate them
- Times must be supplied in the format hh:mm:ss[.ssssss] not all decimal places required
- Datetimes must be supplied in the format yyyy-mm-dd hh:mm:ss[.ssssss]

To evaluate a date, time, or datetime, it needs to be prefaced by the appropriate operator:

- DATE('2010-01-26')
- TIME('23:21:55')
- TIMESTAMP('2010-01-26 23:21:55')

The following key words are supported: **AND, AS, BOTH, CHAR_LENGTH, CHARACTER_LENGTH, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, DATE, DAY, ESCAPE, EXTRACT, FOR, FROM, HOUR, IN, LEADING, LIKE, LOWER, MINUTE, MONTH, NOT, OR, POSITION, SECOND, SUBSTRING, TIME, TIMESTAMP, TRIM, TRAILING, UPPER, YEAR**.

For more information regarding the syntax of SQL92, see the numerous resources on the Internet. The following document is the most complete resource I have found to date: `http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt`.

Working with report1

The report1 type is used as the basis for all three engines. In essence, the other two are variations and enhancements to the core report engine. The quickreport1 and graphicreport1 types each contain a report1 type, but their implementation is primarily about different ways of dealing with the output of the report. It is entirely possible to create other report engine wrappers that handle output in other ways, though it is probably easiest to extend the quickreport1 type to deal with them, since it is already prepared for output to CSV and HTML format as well as sending the output to the clipboard, in addition to print and print preview.

This section will go into some detail regarding the design and usage of the report1 core engine.

The Design of report1

The suite of data types that make up the core report engine consists of:

- report1
- report1aggregate
- report1aggregatevalue
- report1group
- report1groupinst
- report1inst

In practice, these represent three pairs of types: report1 and report1inst, report1aggregate and report1aggregatevalue, and report1group and report1groupinst.

In each case, the first of the pair is used to define the starting information, and the second is used during execution of the report to preserve current state information as it is updated and changes. More on this after we have examined each of the types.

The report1 Type

The type definition is probably the most compact way to look at these types. Here is the type definition of report1:

```
type report1(sqlq1, report1aggregatecontainer) export
  reference
  sqlq1 query resolve  readonly

  embed
  string orderclause   readonly
  boolean distinct     readonly

  event onreportstart
  event onreportend
  event onoutputrow

  dring groups
  dring aggregates

  reference
  type(*) _
  type(*) __ resolve
  type(reportoutputtarget) outputtarget
  SBLNumSettings numlocale
  SBLlocaledateinfo datelocale

  function addgroup         readonly
  function removegroup      readonly
  function addaggregate     readonly
  function removeaggragate  readonly
  function run              readonly
  function setorderclause   readonly
end type
```

Let's start at the beginning. The first thing we see is two type tags: sqlq1, and report1aggregatecontainer. This allows a report1 object to be assigned to any variable that has been declared as able to contain one of these two types. Following that, we see as the first thing, a sqlq1 type parameter called *`query`*, which is marked as `resolve`. Since it is marked that way, all of the properties and methods of the sqlq1 type will appear as part of the report1 type.

Note

Column names are case-sensitive, so when using them in various parts of the report, such as the where or order clause, make sure to use the exact name as specified in the select clause.

The next two items are the orderclause and the distinct properties. The sqlq1 carries out the query, but does not handle the ordering of the output. This is handled by the report1 engine. The name used in the orderclause must match the names used in the columns passed to the select clause. If a field name has its name changed using the `AS` operator, then the name following the `AS` operator must be used in the order clause. Sorting is done in ascending order by default. To reverse it, add the `DESC` key word preceded by a space following the column name. Sorting of text currently does not support any other collation order except native Unicode number, which means that lowercase letters will sort out of sequence with uppercase characters. The second property, distinct, if set ensures that if an entire output line is duplicated, that duplicates do not appear in the output. This can happen under certain circumstances with various filters and joins between tables.

Following on from there, three events are listed: onreportstart, onreportend, and onoutputrow. If defined, these events will be called at the appropriate times, as can be inferred from their names. When implementing some code that makes use of the report engine, at the very least you would want to create a handler for the onoutputrow event. This will get called each time a row is read. The function is passed the following parameters:

1. report1
2. report1inst
3. array of column information
4. array containing the current column values
5. * optional reference if defined for the event

The first two are the report itself and the current running instance. The details about the instance will be found below. The third parameter is a 2-dimensional array that starts at 1, and in the n,0 position is the data type, and in the n,1 position is the display format for that column. The columns are in the same order as when they are passed in to the select clause of the query (these are not necessarily 1:1 with fields, since the select clause allows the use of SQL92 functions to create calculated columns). The number of columns can be retrieved using the report1.`getcolumncount()` method. The fourth parameter is a 1-dimensional array starting at 1 that contains the values for the current row for each of the columns. The values will be of the same data type (or a compatible one) as that of the column.

For completeness, the parameter lists of the other two events are:

1. report1
2. report1inst
3. * optional reference if defined for the event

1. report1
2. report1inst
3. * optional reference if defined for the event

The next parameter is a dring called groups. This contains the ring of report1group objects that will be processed by the report. Groups are processed in the order they are added, so the outermost group will be the first one added, and the innermost group will be the last one added. For example, if you are reporting on name and address data, and grouping on city and then by surname, the city group should be added first, and then the surname group. Also, the sort order should be "city,surname" in order to get the results that are expected.

The aggregates property is also a dring that contains the report-level aggregate values to be computed, each of which is of type report1aggregate. All aggregates only work with numeric columns, except for the count aggregate which is not associated with a column at all (the column number should be set to 0). The supported aggregates currently are: sum, mean, median, mode, and count.

The outputtarget property is not used by the report1 type, since it is not actually concerned with the output at all. It is there to be used by types that deal with output.

The numlocale and datelocale properties should be passed in so as to ensure that the output is formatted correctly. If the application is using the `appframework.sml` library, then the application object will make these available using the exact same types, so that consistency can be assured across the application.

The usage of the methods should be pretty clear from their names. They provide a method for adding and removing groups and aggregates, setting the order clause, and running the report. There will be a large number of additional methods exposed that are part of the sqlq1 type. These include:

- `adddb1table()` – Use this to add database tables to the report
- `setselectclause()` – Call this to set the string representing the select clause
- `setwhereclause()` – This establishes the filter and joins for the report
- `setdefaultformats()` – It is important to add the default formats for the various data types
- `prepare()` – Prepares the report to be run and checks the select and where clauses

The other methods are used while the report is running but are used by the report engine itself, so you shouldn't need to use them unless you are trying to use the sqlq1 type on its own, which is an advanced topic.

The report1aggregate Type

Aggregate values can be calculated at the report or group level. In each case they make use of the same types: report1aggregate and report1aggregateinst. The creation of an aggregate for a qualified column is quite simple, and is done the same for both report-level and group-level aggregates. The key is the first parameter to the `new()` method of the type. Here are the parameters to the method:

1. type(report1aggregatecontainer) *`container`*
2. function *`getvalue`*
3. integer *`colno`*

4. type `datatype`
5. integer `typeid`
6. integer `error`

The first two parameters must be passed, or the creation of the object will fail. In the case of the count aggregate, the *`colno`* parameter is not required (but in the quick report and graphic report versions is set to 0). In all other cases the *`colno`* parameter will also be needed, as will the *`datatype`* parameter. The *`typeid`* parameter is used by both the quick report and graphic report libraries, but is not used by this one. The final parameter is as usual, the *`error`* parameter, and should be a pre-initialized integer in order to get the value back should the object fail to be created.

The only other thing that needs to be done to use the aggregate in the report is to assign the onupdate event handler. Each time a row is read, the aggregate values need to be updated. The function assigned to this event for the specific aggregate handles doing the appropriate type of update.

The report library contains ten functions that are used together with the aggregates, five of them for providing the getval functionality and five for providing the update functionality. These are:

- `report1_agg_getval_count()`
- `report1_agg_update_count()`
- `report1_agg_getval_mean()`
- `report1_agg_update_mean()`
- `report1_agg_getval_median()`
- `report1_agg_update_median()`
- `report1_agg_getval_mode()`
- `report1_agg_update_mode()`
- `report1_agg_getval_sum()`
- `report1_agg_update_sum()`

The report1aggregatevalue Type

The only place that you might encounter this type, is if you decide to implement your own aggregate value type and handler. This type is one of the parameters passed to the getval and update functions of an aggregate implementation. Unless you need to do that, you don't really need to worry about this type. Doing this is an advanced topic.

The report1group Type

In order to provide a grouping functionality within the report, we implemented the report1group type. This type contains the static definition of a group that is used in a report. This includes the two events: ongroupstart and ongroupend, the column number (colno), the name of the group (typically the column name), its data type, and if defined, any aggregate values. Aggregates work exactly the same way as with the report, and use the same type. When adding a group to a report, the `addgroup()` method of the report is called. To add an aggregate to a group, call the `addaggregate()` method of the group.

The report1groupinst Type

This type is only used by event handlers that are dealing with the ongroupstart and ongroupend events. The report1groupinst type contains the current information about this instance of the group, including its value and in the ongroupend event also the various aggregates that may have been defined for the group.

Creating a Report in Source Code

Creating a report is not particularly complicated. Using the address.sbm from the Address Book example (see the SIMPOL Quick Start Guide), a sample report can be seen in the code below. This report outputs a tab-delimited carriage-return and linefeed delimited file of the data from the selected columns.

```
function main()
  report1 report
  sbme1 sbmfile
  sbme1table address
  integer e, erridx
  string s, errmsg
  fsfileoutputstream fpo

  e = 0
  sbmfile =@ sbme1.new("address.sbm", error=e)
  if sbmfile =@= .nul
    s = "Error number " + .tostr(e, 10) + \
        " opening ""address.sbm""{d}{a}"
  else
    address =@ sbmfile.opentable("Address", \
               recordidfieldname="recid_ro_internal", error=e)
    if address =@= .nul
      s = "Error number " + .tostr(e, 10) + " opening the \
           ""Address"" table{d}{a}"
    else
      errmsg = ""
      erridx = 0
      report =@ report1.new()
      report.setselectclause("AddressID, FirstNames, Surname, \
        City, CountryCode", errmsg, erridx)
      report.setwhereclause("", errmsg, erridx)
      report.adddb1table(address)
      report.setorderclause("Surname")
      fpo =@ fsfileoutputstream.new("addresslist.txt", error=e)
      if fpo =@= .nul
        s = "Error number " + .tostr(e, 10) + " opening output \
             file 'addresslist.txt'{d}{a}"
      else
        report.onreportstart.function =@ \
          report1_tabbed_output_reportheader
        report.onreportstart.reference =@ fpo
        report.onoutputrow.function =@ report1_tabbed_output_row
        report.onoutputrow.reference =@ fpo

        report.run(errmsg, erridx, error=e)
        if not (errmsg > "" or e != 0)
```

```
          s = "Success!{d}{a}"
        else
          if errmsg > ""
            s = errmsg + "{d}{a}"
          else
            s = "Error number " + .tostr(e, 10) + " running \
                 report{d}{a}"
          end if
        end if
      end if
    end if
  end if
end function s


function report1_tabbed_output_reportheader(report1 report, \
  report1inst reportinst, fsfileoutputstream fpo)

  integer cnt, i
  string title, outline, emsg

  emsg = ""
  outline = ""
  cnt = report.getcolumncount()
  i = 1
  while i <= cnt
    title = report.getcolumntitle(i, emsg)
    if title > ""
      outline = outline + .if(i > 1, '{9}', '') + title
    else
      outline = outline + .if(i > 1, '{9}', '') + ""
    end if
    i = i + 1
  end while

  outline = outline + "{d}{a}"
  fpo.putstring(outline, 1)
end function


function report1_tabbed_output_row(report1 report, report1inst \
  reportinst, array columns, array currcolvals, \
  fsfileoutputstream fpo)

  integer cnt, i
  anyvalue value
  string svalue
  string outline
  string displayformat
  type datatype

  outline = ""
  value =@ anyvalue.new()
  cnt = report.getcolumncount()
  i = 1
  while i <= cnt
    value = currcolvals[i]
    datatype =@ columns[i,0]
    displayformat = columns[i,1]
    svalue = val2string(datatype, value, report.datelocale, \
               report.numlocale, displayformat, .false)
    if svalue > ""
      outline = outline + .if(i > 1, '{9}', '') + svalue
    else
      outline = outline + .if(i > 1, '{9}', '') + ""
    end if
    i = i + 1
  end while

  outline = outline + "{d}{a}"
  fpo.putstring(outline, 1)
end function
```

The previous sample program demonstrates the use of two events to handle the initial output of the header, and then to output the data for each row. It also sorts the results according to the `Surname` column. As can be seen from the source code, there isn't much required to create a report using code, especially once the event handlers have been written. The two event handlers here are not specific to the data, so they can be used to output any result in tab-delimited format.

report1 Summary

In this section we have learned about the design of and how to work with the report1 type. We have also discovered that although it doesn't take much code to create a report this way, that it doesn't actual produce output unless we write it ourselves. In the next two sections, we will have no more effort, but we can get output to a window or the printer.

Working with quickreport1

Working with the quickreport1 type is similar to working with the report1 discussed in the previous section. One of the main differences is that this report handles output to various targets, and therefore needs to know more about the content. It also has the concept of a title, page numbering, and showing the current date at the top of each page (all optional), plus displaying column headings and coping with columns where the data is too long. It is limited to one font that it uses for the entire report and has the advantage that it is quite simple to define. The Quick Report also supports grouping, sorting, and output of group and report aggregate values, such as the count of rows plus the sum, mean, medium, or mode for a column in the report and groups. To begin, let's have a look at the definition of the quickreport1 type:

```
type quickreport1 export
  embed
  boolean dirty

  integer outputtarget
  boolean valid
  string filename
  event onpagechange
  event onoutputheader
  event onoutputfooter
  event onbeforerow
  event onafterrow
  event onbeforegroup
  event onaftergroup
  event onoutputreportheader
  event onoutputreportfooter

  // flag indicating if the report header has been output yet,
  // so people can suppress the page header output
  boolean reportheaderoutput
  // allows people to suppress row output and just output
  // totals of groups or the whole report
  boolean suppressrowoutput

  // Properties for output to window or printer
  integer paperwidth
  integer paperheight
  integer marginleft
  integer margintop
  integer marginright
  integer marginbottom
  integer dpix
  integer dpiy
  boolean showpagenumber
  boolean showdate
  boolean showtitle
  string title
  string dialogdata
  number wrapcharcountkludgevalue
  integer currpagenumber
  integer currrownumber
  integer currtopofpage
  integer rowheight
  number rowheightadjustment    readonly
  integer lastreportedpagenumber
  // used to reserve an area for a footer,
  // to throw an early end of page
  integer footerlinecount

  string defnumberformat
  string defdateformat
  string deftimeformat
  string defdatetimeformat
  string defintegerformat
  string defbooleanformat

  dring columninfo
  dring datasources
  dring tables

  boolean usegauge
  gaugedialog gauge reference
```

```
reference
wxfont italicfont
wxfont underlinefont
wxfont bolditalicfont
wxfont boldunderlinefont
wxfont italicunderlinefont
wxfont bolditalicunderlinefont
wxfont headerfont
wxfont pagefont
wxfont origfont
report1 report resolve

// The following are for output types 1 and 2 (window and printer)
wxprintout printout
wxprintpagetemplate currtemplate
wxprintpage currpage
GDI gdi
WINSPOOL winspool

embed
number fontwidthratio
number fontheightratio
boolean usewrapheight2

// This is only for output type 1
boolean centeroverdisplay
boolean startat100percent

// For clipboard output
boolean suppressoutputmessages

reference
// This is for the clipboard target (tab separated and crlf separated)
array clipoutput

// This one is for the HTML output
fsfileoutputstream fpo
string outputfilename embed
string stylefilename embed
boolean tbodyoutput embed
boolean outputrowodd embed

// This is for the CSV output
dbQRImport qrimportconverter
dbCSVExport csvexportconverter
boolean headeroutput embed

// And this is for the SBME output
dbSBMEExport sbmeexportconverter
string targettablename embed

array columns; // This is assigned after a callback has happened from
               // the report engine, it contains the column information
```

```
  function addaggregate           readonly
  function addcolumninfo          readonly
  function adddatasource          readonly
  function addtable               readonly
  function finddatasource         readonly
  function findtable              readonly
  function getwrapheight          readonly
  function getwrapheight2         readonly
  function getprinttextextent     readonly
  function run                    readonly
  function outputextraline        readonly
  function setrowheightadjustment readonly

  function addgroup               readonly

  function getcolumninfobycolno   readonly
end type
```

As we can see, this type definition is considerably more complex than the one for the report1 type. In fact, if you look closely you will find that it actually contains the report1 type in addition to all of its extensions. The good news is, you don't need to worry about most of it, since it just works. The important bits to be aware of are the paperwidth, paperheight, and margin properties. Virtually everything is handled in the call to the quickreport1.`new()` method. Below is some sample code that demonstrates how to create a Quick Report:

```
include "quickreporthdr.sma"

function main()
  integer e, erridx
  string s, errmsg
  sbme1 sbmfile
  sbme1table address
  quickreport1 qr
  wxfont font
  report1group group
  quickreport1datasource ds1

  e = 0
  sbmfile =@ sbme1.new("address.sbm", error=e)
  if sbmfile =@= .nul
    s = "Error number " + .tostr(e, 10) + " opening \
         ""address.sbm""{d}{a}"
  else
    address =@ sbmfile.opentable("Address", \
       recordidfieldname="recid_ro_internal", error=e)

    if address =@= .nul
      s = "Error number " + .tostr(e, 10) + " opening the \
           ""Address"" table{d}{a}"
    else
      errmsg = ""
      erridx = 0
      font =@ wxfont.new("Arial Narrow", 10, "n", "n", "", \
                         error=e)
      qr =@ quickreport1.new(outputtarget=QR_OUTPUTWINDOW, \
               title="Address List", pagefont=font, error=e)
      qr.setselectclause("AddressID, FirstNames, Surname, \
                         City, CountryCode", errmsg, erridx)
      qr.setwhereclause("", errmsg, erridx)
      ds1 =@ qr.adddatasource(sbmfile.type, "address.sbm", \
                              sbmfile, error=e)
      qr.addtable(address, ds1, error=e)
      qr.setorderclause("City, Surname")
      group =@ qr.addgroup("City", 4, string, error=e)
      if group !@= .nul
        qr.addaggregate(group, QR_AGG_COUNT, .nul, integer, \
                        error=e)
      end if

      qr.addcolumninfo( 20000,  12000, "right,top", error=e)
      qr.addcolumninfo( 34000,  30000, error=e)
      qr.addcolumninfo( 66000,  40000, error=e)
      qr.addcolumninfo(108000,  65000, error=e)
      qr.addcolumninfo(175000,  6000,  error=e)
      qr.showdate = .true
      qr.showpagenumber = .true
      qr.showtitle = .true
      qr.usewrapheight2 = .true
      qr.addaggregate(.nul, QR_AGG_COUNT, .nul, integer, error=e)

      // The following commented out lines show how to save
      // and load a Quick Report using the XML format
      //savequickreport(qr, "addresslist.sxq", error=e)
      //qr =@ loadquickreport("addresslist.sxq", error=e, \
      //      errortext=errmsg)

      qr.startat100percent = .true
      qr.centeroverdisplay = .true

      qr.run(errmsg, erridx, error=e)

      if not (errmsg > "" or e != 0)
        wxprocess(20000000)
        s = "Success!{d}{a}
      else
        if errmsg > ""
          s = errmsg + "{d}{a}"
        else
          s = "Error number " + .tostr(e, 10) + \
              " running report{d}{a}"
        end if
      end if
    end if
  end if
end function s
```

The preceding program code should be fairly self-explanatory, but we will go through it briefly touching on the interesting points. In this program we decided to use `Arial Narrow 10 points` for our report. The font is created first and passed to the `new()` method. This is *important*! For various reasons, the Quick Report code makes variants of the font, so it is necessary to pass in the font when the object is created. There is no provision for changing it later. The outputtarget should be one of the valid output targets. There are six different valid target types, including: window, printer, CSV file, HTML file, clipboard, and database (SBME). The HTML target is similar to the window and printer targets, in that it produces a formatted report, albeit in one long page. The other three targets are well-suited to exporting data in their respective formats (clipboard produces a tab separated, newline separated output that can be directly pasted into programs like MS Excel). In the case of the data export targets, the aggregates, grouping, and other formatting information is ignored. The various constants for output and aggregate types can be found in the `quickreporthdr.sma`.

Once the quickreport1 object has been created, the select clause, where clause, and order clause are defined, and the data source and tables are added (just as in report1). Then we add a group, and an aggregate for the group. Finally we define the column information. This is required to be defined in the same order as the list of columns in the select clause. The `addcolumninfo()` method takes the following parameters:

1. integer *`columnstart`*
2. integer *`columnwidth`*
3. string *`alignment`*
4. boolean *`wrap`*
5. integer *`error`*

The first two parameters are the horizontal starting position (from the left edge of the paper – the left and right margin values are ignored currently) and the width of the column. These are measured in micrometers (millionths of a meter). The next is the alignment. This can be one of:

- `"left,top"` (the default)
- `"top"` (centered horizontally)
- `"right,top"`
- `"left"` (centered vertically)
- `""` (centered vertically and horizontally)
- `"right"` (centered vertically)
- `"left,bottom"`
- `"bottom"` (centered horizontally)
- `"right,bottom"`

The fourth parameter is *`wrap`*. This will attempt to ensure that the content is wrapped around within the confines of the column and extends the line down the page until the content has been output. If the content is too large to fit on the page, it will be moved to the next page. If it is too large to fit on a page, it will be truncated. Using this feature will slow down the report, since for every row the content of each column with this feature will need to be tested to see if it fits and if not, how much space it requires. Also, there are two algorithms, one is much faster than the other, but is less precise. To use the faster algorithm, as shown in the preceding sample, the usewrapheight2 property must be set to `.true`.

Once the column information has been added, the program sets the switches to `.true` to enable the page title to be output, the date, and the page count. It then adds an aggregate for the overall count of rows in the report. Following that are two commented out program statements, that demonstrate how to save the report to disk, and how to load the report from disk. The default file extension for SIMPOL Quick Reports is `.sxq`. Finally, there are two options, one called startat100percent and centeroverdisplay, both of which currently only work on Windows (they are implemented by calling functions in the Windows API that do not have equivalents elsewhere). Once everything is prepared, the `run()` method is called. Since this program is just a sample, it then enters a `wxprocess()` call for 20 seconds, so that there is time to look at the output in the window. Without this call, the program would simply exit and the output window would immediately close. In a normal application that has a window open and which is sitting in a `wxprocess()` loop anyway, this would not be an issue.

Enhanced Quick Report Output

With the 1.8 release of SIMPOL Professional, a number of new capabilities were added. There are now the following events for formatting output to the print preview window or the printer:

- onoutputheader – Called after the title, date, and page number line is output (if it is output) and before the column titles are output. The prototype for a function that handles this event is: `funcname` (quickreport1 *`quickreport`*, report1 *`report`*, report1inst *`reportinst`*, type(*) *`reference`*).
- onoutputfooter – Called just before calling the function `report1_quickreport_outputpageheader()`. There is no standard footer output by the Quick Report. The prototype for a function that handles this event is: `funcname` (quickreport1 *`quickreport`*, report1 *`report`*, report1inst *`reportinst`*, type(*) *`reference`*).
- onbeforerow – Called just before calling the code that outputs the row data. This call is not suppressed even if the property supressrowoutput is set to `.true`. The prototype for a function that handles this event is: `funcname` (quickreport1 *`quickreport`*, report1 *`report`*, report1inst *`reportinst`*, type(*) *`reference`*).
- onafterrow – Called just after calling the code that outputs the row data. This call is not suppressed even if the property supressrowoutput is set to `.true`. The prototype for a function that handles this event is: `funcname` (quickreport1 *`quickreport`*, report1 *`report`*, report1inst *`reportinst`*, array *`columns`*, array *`currcolvals`*, type(*) *`reference`*).
- onbeforegroup – Called as the group changes. No group header is output by the Quick Report, so the user can use this as they wish. The prototype for a function that handles this event is: `funcname` (quickreport1 *`quickreport`*, report1group *`group`*, report1groupinst *`groupinst`*, type(*) *`reference`*).
- onaftergroup – Called just after calling the code that outputs the the aggregate values and the count (if any are defined). The prototype for a function that handles this event is: `funcname` (quickreport1 *`quickreport`*, report1group *`group`*, report1groupinst *`groupinst`*, type(*) *`reference`*).
- onoutputreportheader – Called just after calling the code that outputs the page header, including the columns headings. This may be modified in a later version, or as an alternative, a method of suppressing the column headings at the start of the report may be added. The prototype for a function that handles this event is: `funcname` (quickreport1 *`quickreport`*, report1 *`report`*, report1inst *`reportinst`*, type(*) *`reference`*).
- onoutputreportfooter – Called just after calling the code that outputs the the aggregate values and the count (if any are defined). No page footer is currently output on the final page. This may be corrected in a future release. If it is, it will be made optional. The prototype for a function that handles this event is: `funcname` (quickreport1 *`quickreport`*, report1 *`report`*, report1inst *`reportinst`*, type(*) *`reference`*).

In all of these events, to output anything to the window or printer, it is necessary to create an instance of a quickreportextraoutputinfo type. This contains all of the information required to print this content, including the position, alignment, name (this is the name given to the template item and must be different from the printname), printname (this is the name given to the string data and if not supplied it will be derived from the name), and the text (the actual string data that should be output).

This is passed to the `outputextraline` method of the quickreport1 object. The prototype for this function is: quickreport1.`outputextraline` (quickreport1 *`me`*, quickreportextraoutputinfo *`outputinfo`*, string *`fontcharacteristics=""`*, boolean *`incrementtopofpage=.true`*, integer *`error`*). The *`fontcharacteristics`* should contain either nothing, or any of: `biu`, `bi`, `bu`, `iu`, `i`, `u`, or `b`, where `b` is bold, `i` is underline, and `i` is italic.

By setting the value of the *`incrementtopofpage`* parameter to `.false` it is possible to output multiple values on the same line. It is important that in the final call to this function (if making multiple calls) that this parameter is set to `.true`.

quickreport1 Summarizing Quick Report Output

Another new feature added in the 1.8 release of SIMPOL Professional was the suppressrowoutput of the quickreport1 type. If this is set to `.true`, then only the report and group aggregate values will be output. This feature is only available to the window and printer output targets.

quickreport1 Summary

As we have discovered in this section, creating a Quick Report in code is not very difficult. It is similar to the code used for a report1 report, but with the extra requirement of selecting a font and defining the column information. It is quick and easy, but it has the down side that everything on the report is in one font, and there is little flexibility over the layout and none regarding group headers and footers. This latest version does support some additional enhancements, such as summarizing the output by suppressing the row output, and offers the new events and the new method `outputextraline()` which can be used to output additional content. For more complex reports, where all of the items in headers and footers can be defined, and images can be incorporated, we need to use the Graphic Report, which is the subject of the next section.

Working with graphicreport1

The Graphic Report is extremely flexible in its design. The actual report is separate from the physical representation of its output. The graphicreport1 type contains a graphicreport1form type, which in turn contains a dring of graphicreport1formpage objects. The graphicreport1 type also incorporates the report1 type and is therefore similar to working with report1 as discussed previously. Unlike the quickreport1 type, however, this type is considerably more powerful and therefore also considerably more complex. It uses the printform1 type to provide templates for each area of the report. Each band of the report is defined as a page in the graphicreport1form. See Chapter 24, *Using Data-Aware Print Forms in SIMPOL* for more information about printform1.

The Graphic Report is a banded report system. That means that the components are broken up into bands, each the width of the page. The bands that are provided include:

- Page header
- Page footer
- Body

- Report header
- Report footer
- Group header
- Group footer

None, some or all of these bands may be used (the group bands are for each group defined). When the report is run, it assembles the page from these bands. At the start of the report it will output the report header if it has been defined and at the beginning of each page it outputs the page header if that has been defined. Since it may be messy to have both on the same page, there is an option to suppress output of the page header on the first page. Following that, if there are any groups defined and there is a group header for the group, that will be output, then the body section of the report will be output until a group change, or the bottom of the page, (allowing for the page footer if it has been defined). Each page is then assembled as required until the end of the report is reached, at which point the report footer will be output followed by the page footer (this can also be suppressed on the last page). Each band is defined as a graphicreport1formpage that contains a printform1page object. Each of these pages has a specific naming format so that the engine will recognize them. These are also stored as constants in an includable SIMPOL source code file called `graphicreporthdr.sma`, but the list is as follows:

- `"pageheader"`
- `"pagefooter"`
- `"body"`
- `"reportheader"`
- `"reportfooter"`
- `"groupheader"`
- `"groupfooter"`

In turn, each page can contain any of the following content elements:

- graphicreport1arc
- graphicreport1ellipse
- graphicreport1line
- graphicreport1rectangle
- graphicreport1triangle
- graphicreport1formtext
- graphicreport1formbitmap

Each of these contains a printform1 graphic or control of equivalent type. In the case of the graphics, there is little difference between them. The bitmap and text objects are different however, since they can be associated with a column value in the body page. In addition, the text objects can also be associated with an aggregate value in the group and report footers, or defined as a calculation using a system variable in the group header (for the GROUP items below), otherwise any of them can be used anywhere, though page headers and footers would be the most logical choice for most. The supported system variables include:

- PAGE – (returns the page number in the report formatted using the minimum characters (pure `.tostr()` call
- TODAY – (returns the current date formatted using the default date format and date locale information as provided to the report
- NOW – (returns the current time formatted using the default time format information as provided to the report
- TIMESTAMP – (returns the current date and time formatted using the default datetime format and date locale information as provided to the report
- COUNT – (returns either the count of rows in the report or the group, depending on the page)
- GROUP – (returns the value of the current group)
- GROUPNAME – (returns the name of the column for the current group)
- GROUPINFO – (returns the name of the current group, followed by a colon and a space, and then the group's current value

Each item is placed on the page using print coordinates (to the nearest micrometer). Positioning is absolute, so if something is too close to an edge to be printed without being cropped, then it *will* be cropped.

When creating any of these controls, one of the arguments to each is an appropriate printform1control or printform1graphic object. In the case of the two form objects, they can each take a column number (which is based on the order of the select clause), in the *`colno`* parameter. Furthermore, the text control can also be assigned an aggregate type, or instead of a column number it can have a calculation assigned. Both of the form controls can also be assigned static values, a fixed bitmap or text value. All of this is handled in the graphicreport1form.`addcontrol()` method.

The easiest way to understand how to use the report is to make one:

```
            include "graphicreporthdr.sma"

function main()
  integer e, erridx, stdtexthgt
  string s, errmsg
  sbme1 sbmfile
  sbme1table address
  graphicreport1 gr
  wxfont font, font2, font3, font4
  report1group group
  dataform1datasource ds1
  graphicreport1formpage page
  graphicreport1formtext ptxt
  SBLlocaledateinfo datelocale
  SBLNumSettings numlocale

  e = 0
  sbmfile =@ sbme1.new("address.sbm", error=e)
  if sbmfile =@= .nul
    s = "Error number " + .tostr(e, 10) + " opening \
         ""address.sbm""{d}{a}"
  else
```

```
    address =@ sbmfile.opentable("Address", \
       recordidfieldname="recid_ro_internal", error=e)

    if address =@= .nul
      s = "Error number " + .tostr(e, 10) + " opening the \
           ""Address"" table{d}{a}"
    else
      datelocale =@ SBLlocaledateinfo.new(format="dd/mm/yy")
      numlocale =@ SBLNumSettings.new("£", ",", ".", .false)
      stdtexthgt = 4900
      errmsg = ""
      erridx = 0
      font =@ wxfont.new("Arial Narrow", 10, "n", "n", "", error=e)
      font2 =@ wxfont.new("Arial Narrow", 10, "n", "b", "",error=e)
      font4 =@ wxfont.new("Arial Narrow", 13, "n", "b", "",error=e)
      font3 =@ wxfont.new("Arial", 14, "n", "b", "", error=e)
      gr =@ graphicreport1.new(paperwidth=210000, \
            paperheight=297000, outputtarget=GR_OUTPUTWINDOW, \
            title="Address List", datelocale=datelocale, \
            numlocale=numlocale, error=e)
      gr.reportform.wrapkludgevalue = .toval("1.15", .nul, 10)
      gr.reportform.fontresizekludgevalue = .toval("0.7", .nul, 10)
      gr.reportform.wrapcharcountkludgevalue = .toval("1", .nul,10)
      gr.usewrapheight2 = .true

      gr.setselectclause("AddressID, FirstNames, Surname, City, \
                          CountryCode", errmsg, erridx)
      gr.setwhereclause("", errmsg, erridx)
      gr.setorderclause("City, Surname")
      ds1 =@ gr.adddatasource(sbmfile, "address.sbm", error=e)
      gr.addtable(address, ds1, error=e)

      // Body Page
      page =@ gr.addpage(210000, 600 + stdtexthgt, 0xffffff, \
              name=sGR_BODY, error=e)
      gr.addcontrol(graphicreport1formtext, printleft=20000, \
            printtop=300, printwidth=12000, \
            printheight=stdtexthgt, printalignment="right,top", \
            font=font, printname="tbAddressID", page=page, \
            colno=1, error=e)
      gr.addcontrol(graphicreport1formtext, printleft=34000, \
            printtop=300, printwidth=50000, \
            printheight=stdtexthgt, font=font, \
            printname="tbFirstNames", page=page, colno=2, error=e)
      gr.addcontrol(graphicreport1formtext, printleft=86000, \
            printtop=300, printwidth=50000, \
            printheight=stdtexthgt, font=font, \
            printname="tbSurname", page=page, colno=3, error=e)
      gr.addcontrol(graphicreport1formtext, printleft=138000, \
            printtop=300, printwidth=50000, \
            printheight=stdtexthgt, font=font, \
            printname="tbCity", page=page, colno=4, error=e)
      gr.addcontrol(graphicreport1formtext, printleft=190000, \
            printtop=300, printwidth=12000, \
            printheight=stdtexthgt, font=font, \
            printname="tbCountryCode", page=page, colno=5, error=e)

      // Page Header
      page =@ gr.addpage(210000, 16320 + stdtexthgt, 0xffffff, \
            name=sGR_PAGEHEADER, error=e)
      gr.addcontrol(graphicreport1formtext, printleft=50000, \
            printtop=6000, printwidth=110000, printheight=8200, \
            printalignment="", text="Address List", font=font3, \
            printname="lPageTitle", page=page, error=e)
      gr.addcontrol(graphicreport1formtext, printleft=20000, \
            printtop=16000, printwidth=12000, \
            printheight=stdtexthgt, printalignment="right,top", \
            text="Addr ID", font=font2, printname="lAddressID", \
            page=page, error=e)
      gr.addcontrol(graphicreport1formtext, printleft=34000, \
            printtop=16000, printwidth=50000, \
            printheight=stdtexthgt, text="First Names", \
            font=font2, printname="lFirstNames", page=page, \
            error=e)
      gr.addcontrol(graphicreport1formtext, printleft=86000, \
            printtop=16000, printwidth=50000, \
            printheight=stdtexthgt, text="Surname", font=font2, \
            printname="lSurname", page=page, error=e)
      gr.addcontrol(graphicreport1formtext, printleft=138000, \
            printtop=16000, printwidth=50000, \
            printheight=stdtexthgt, text="City", font=font2, \
            printname="lCity", page=page, error=e)
      gr.addcontrol(graphicreport1formtext, printleft=190000, \
            printtop=16000, printwidth=12000, \
            printheight=stdtexthgt, text="Ctry", font=font2, \
            printname="lCountryCode", page=page, error=e)
      gr.addgraphic(graphicreport1line, point.new(20000, \
            16300 + stdtexthgt), point.new(202000, 16300 + \
            stdtexthgt), width=100, printname="lBorder", page=page, \
            error=e)

      // Page Footer
      page =@ gr.addpage(210000, 9000 + STDTEXTHGT, 0xffffff, \
            name=sGR_PAGEFOOTER, error=e)
      gr.addgraphic(graphicreport1line, point.new(20000, 1000), \
            point.new(190000, 1000), width=100, \
            printname="lBorderFooter", page=page, error=e)
      ptxt =@ gr.addcontrol(graphicreport1formtext, \
            printleft=98000, printtop=3000, printwidth=14000, \
            printheight=stdtexthgt, printalignment="", text="", \
            font=font2, printname="lPageNo", page=page, error=e)
      if ptxt !@= .nul
        ptxt.calculation = "PAGE"
      end if

      group =@ gr.addgroup("City", 4, string, error=e)
      if group !@= .nul
        gr.addaggregate(group, GR_AGG_COUNT, .nul, integer,error=e)
```

```
   end if

   // Group Header
   page =@ gr.addpage(210000, 12000, 0xffffff, \
         name=sGR_GROUPHEADER, group=group, error=e)
   if page !@= .nul
     ptxt =@ gr.addcontrol(graphicreport1formtext, \
             printleft=20000, printtop=5000, printwidth=30000, \
             printheight=integer.new(stdtexthgt * (135/100)), \
             printalignment="left,top", text="", font=font4, \
             printname="lGroupname", page=page, error=e)
     if ptxt !@= .nul
       ptxt.calculation = "GROUPINFO"
     end if
   end if

   // Group Footer
   page =@ gr.addpage(210000, 8500, 0xffffff, \
           name=sGR_GROUPFOOTER, group=group, error=e)
   if page !@= .nul
     ptxt =@ gr.addcontrol(graphicreport1formtext, \
         printleft=20000, printtop=2000, printwidth=30000, \
         printheight=integer.new(stdtexthgt * (135/100)), \
         printalignment="left,top", text="", font=font4, \
         printname="lGroupcount", page=page, error=e)
     if ptxt !@= .nul
       ptxt.calculation = "COUNT entries"
     end if
   end if

   gr.addaggregate(.nul, GR_AGG_COUNT, .nul, integer, error=e)

//     savegraphicreport(gr, "addresslist.sxr", error=e)
//     gr =@ loadgraphicreport("addresslist.sxr", error=e, \\
//           errortext=errmsg)

   gr.startat100percent = .true
   gr.centeroverdisplay = .true

   e = 0
   gr.run(errmsg, erridx, error=e)
   if not (errmsg > "" or e != 0)
     wxprocess(20000000)
     s = "Success!{d}{a}"
   else
     if errmsg > ""
       s = errmsg + "{d}{a}"
     else
       s = "Error number " + .tostr(e, 10) + \
           " running report{d}{a}"
     end if
   end if
  end if
 end if
end function s
```

As can be seen from the preceding code, there is a lot more involved in creating a Graphic Report than there is for a Quick Report, but the difference is in the amount of control over the resulting look of the report. Just as with the Quick Report, the initial stages of creating a Graphic Report consists of opening the data source(s) and table(s), creating the graphicreport1 object, and setting the select, where, and order clauses. In addition, the fonts that will be used are created, and there is a set of properties that are related to the "wrap" functionality that can be set. These occasionally need tweaking to get the best results. Like with the Quick Report, only add the "wrap" capability if it is required, since it adds considerable processing overhead to each time a control that uses it is output. Two of the arguments passed to the `new()` method of the graphicreport1 are the *`pagewidth`* and the *`pageheight`* parameters. The ones used in the example are for A4 paper. The US Letter paper size is: 215900 x 279400.

Once the standard tasks have been dealt with, the various page bands are added, each with the controls that are required. Interestingly, just because a column is in the select statement does not mean that it will appear in the report. Unless it is associated with a control in the body page, there will no output for that column. This is useful when it is necessary to retrieve extra column information for use in report or group footers. Also, to do a summarization, it is only necessary to not define the body page.

If the level of control that is available in the basic design is still not enough, at the point of outputting a page chunk onto the final output page, there is an onoutput event for each page that can be used to call the programmer's code. A different function should be assigned for each unique band or graphicreport1formpage object. The various function prototypes for the types of pages are as follows.

Table 25.1. onoutput Function Prototypes

Band Type	**Function Prototype**

Body band	**`onoutput_handler`**`(`*`page`*`, `*`pagechunk`*`, `*`report`*`, `*`reportinst`*`, `*`columns`*`, `*`currcolvalues`*`, `*`reference`*`);`
	
	`graphicreport1formpage `*`page`*`;`
	`printform1page `*`pagechunk`*`;`
	`report1 `*`report`*`;`
	`report1inst `*`reportinst`*`;`
	`array `*`columns`*`;`
	`array `*`currcolvalues`*`;`
	`type(*) `*`reference`*`;`

Page header and footer, report header and footer	**`onoutput_handler`**`(`*`page`*`, `*`pagechunk`*`, `*`report`*`, `*`reportinst`*`, `*`reference`*`);`
	
	`graphicreport1formpage `*`page`*`;`
	`printform1page `*`pagechunk`*`;`
	`report1 `*`report`*`;`
	`report1inst `*`reportinst`*`;`
	`type(*) `*`reference`*`;`

Group header and footer	**`onoutput_handler`**`(`*`page`*`, `*`pagechunk`*`, `*`group`*`, `*`groupinst`*`, `*`reference`*`);`
	
	`graphicreport1formpage `*`page`*`;`
	`printform1page `*`pagechunk`*`;`
	`report1group `*`group`*`;`
	`report1groupinst `*`groupinst`*`;`
	`type(*) `*`reference`*`;`

At the point where this is called, all of the data and calculations have been done, and the resulting output can still be manipulated by the programmer. For example, in the body, if the total of a row is negative, the foreground color could be changed to red. This change only affects the current output chunk, not the template, so it needn't be reversed for rows that are positive. The names of the controls on the pagechunk will be the same as on the template, making it easy to address thae various controls. As was the case with the Quick Report example, there are two commented lines of code that save and then load the Graphic Report. The default file extension for SIMPOL Graphic Reports is `.sxr`.

graphicreport1 Summary

In this section we have learned about the use of the graphicreport1 type. As we have discovered, these are much more powerful than any of the other methods of reporting we have previously explored, but the price is that they are more complex to create. When choosing which method to use, it is best to remember that each has its strengths and weaknesses. The basic report1 is meant for specialized purposes such as implementing a custom report style that may not even generate output to the screen. The quickreport1 type is useful for creating reports to window or printer (and eventually other targets), with few more requirements than those of the basic report1 aside from defining column information and which also allows grouping and aggregate calculations. Finally, if the report requires a great deal of control in the final result or needs to incorporate images (or a company letterhead), then the graphicreport1 type is the best report for the job.

Chapter 26. Using the SIMPOL Application Framework

This chapter will describe the general design of the SIMPOL Application Framework. The `appframework.sml` library provides a complete application framework that uses `databaseforms.sml`, `formlib.sml`, `db1util.sml`, `uisyshelp.sml`, and other libraries to allow the quick and easy creation of powerful database-oriented applications. It implements an application and appwindow data type, which together with various functions and helper types, assists the programmer to produce a reliable database-oriented application with very little effort. Typically a program based on the framework will create its own application object type that includes the application object from the framework, and which is type-tagged as `application`. To see the code for this in detail, look at the chapters in the Quickstart Guide that cover the Address Book and Ordering System samples. These samples are also included as part of the distribution, so all the source code is there to exploit.

The Design of the Application Framework

The basic premise behind the application framework is that most applications that work with a database will have numerous aspects in common. These include:

- Displaying forms
- Switching forms
- Managing data sources
- Managing tables
- Creating, locking, unlocking, modifying, and deleting of records
- Placing the cursor in the first viable control on a form when entering data-entry
- Testing to see if a record has been modified, and prompting the user to save or discard it
- Browsing through records

To that end, the application framework provided with SIMPOL does an excellent job. It contains a basic application type, that can be used as is, or which can be embedded into a more sophisticated application type. The appwindow type provides the necessary management for the one or more windows used by the application, provides a container for the database tables used by the window, and various services for setting the current table, record, index, etc. for the window.

This library includes the `formlib.sml`, and therefore has all of the features of that library including all of the lower level included libraries. This includes all of the dataform1, printform1, and DOM type families, plus most of the important functions for lists, formatting types as strings, and the support libraries for the UI and the file system. In this section we will specifically discuss the features specific to the `appframework.sml`.

Table 26.1. The Functions in the Application Framework

Function name	Description
checkneedsave()	Checks to see if the record has been modified. If it has, it prompts the user to save or discard and handles the result. After calling this function, the programmer must check the return value. If it is `.true`, they can continue whatever they are doing that would lose the current unsaved changes, otherwise they should abort the operation.
clearstatusbar()	This clears the status bar after a specified delay (has a default value).
closewindow()	This function is intended to be attached to the onvisibilitychange event of the embedded wxwindow object in an appwindow type. To handle the closing of

Function name	Description
	the final window the programmer creates an event handler for the onexitrequest event of the application. If defined, it will be called from this function if the user is attempting to close the last visible window. Return .false to prevent the window closing.
defer()	This function is typically called from any menu item or tool bar item event handler, in order to ensure that any changed data in the current control with focus has been written to the underlying field. The defer mechanism is necessary since menu and tool bar events take place before the onlostfocus event of a form control. Without the defer mechanism, the call to save changes to a record would fail to record any changes to the control that currently has focus. By using defer, the focus is cleared and the onlostfocus event is called to write the data while the `defer()` function in a separate thread waits a very short period and then re-calls the original function.
deleterecord()	This function is designed to be assigned to both menu items and tool bar items as the onselect event handler. It expects the application object to be passed (or the derived application object as long as it is tagged application). It can also be called directly passing the appwindow object as the first argument. It will handle locking and deletion of the current record on the form, and will call the ondelete event handler if it has been assigned. This function is particularly tuned to cope with both auto-locking and explicit locking systems. It will use the status bar, if available, when telling the user if a record is unable to be locked, which is very important since in an auto-locking environment, when the dialog window vanishes it automatically attempts to place focus back on the form, which then tries to lock the record, leading to an uninterruptible cycle that will only frustrate the users.
duplicaterecord()	This function is designed to be assigned to both menu items and tool bar items as the onselect event handler. It expects the application object to be passed (or the derived application object as long as it is tagged application). It can also be called directly passing the appwindow object as the first argument. It creates a new record and copies the current record into it. To make changes for unique indexes, assign a handler function to the onchangerecord event and test the record to see if it has been stored. If not, then it is a new record that has been duplicated.
findfirstfocusablecon-trol()	This function does exactly what it says. It retrieves the first focusable control on a form. This would be used in conjunction with placing the user in data-entry, in order to set focus to the first appropriate control in the tab order.
getappwindowfromwin-dow()	Use this function to return the appwindow object from a wxwindow object. This situation occurs when a menu event takes place, since the object passed to the function is of type wxmenuitem, and not appwindow. This function needs to be paired with the next one.
getmenuitemwindow()	With this function it is possible to retrieve the wxwindow object from a wx-menuitem object. When a menu event occurs, the object that causes the event is an item in the menu, so that is the type passed as the initial parameter to the function. Together with the previous function you can retrieve both the wxwindow and the appwindow objects.
gettableformatstrings()	Use this function to retrieve an array of the field display format strings for a table that is part of the appwindow ring of tables. The array is in the order of the fields in the database table.

Function name	**Description**
gettablesarray()	Retrieves an array of tbinfo objects, one for each table opened in the appwindow's ring of tables. These objects are not specific to a parent object and can be used for transferring a set of tables from one component to another, so that both sections use the exact same set of objects. That is important so that any record objects selected are compatible to each other.
lookup()	Call this function to look up a value in a database table against a specific index and if found, to return the record. If no matching record is found, then the return value is `.nul`.
modifyrecord()	This function is designed to be assigned to both menu items and tool bar items as the onselect event handler. It expects the application object to be passed (or the derived application object as long as it is tagged application). It can also be called directly passing the appwindow object as the first argument. It will lock and modify the record that is currently displayed in the form. It will also place the user into data-entry in the first focusable control on the form.
newrecord()	This function is designed to be assigned to both menu items and tool bar items as the onselect event handler. It expects the application object to be passed (or the derived application object as long as it is tagged application). It can also be called directly passing the appwindow object as the first argument. It will create a new record for the master table of the currently displayed form. It will also place the user into data-entry in the first focusable control on the form.
saverecord()	This function is designed to be assigned to both menu items and tool bar items as the onselect event handler. It expects the application object to be passed (or the derived application object as long as it is tagged application). It can also be called directly passing the appwindow object as the first argument. It will save the record correctly and reset various state flags. It is important that this function be used for saving the record if the program is calling either the `newrecord()` or `modifyrecord()` functions..

Working with `appframework.sml`

The framework includes all the basic features that are required to produce a working, distributable, database-oriented program that just needs a menu, a tool bar (both of which are available in the samples), plus the code that is specific to that program's functionality (switching forms, calculating field values, running reports, etc.). It also includes support for both approaches to the user-interface, auto-locking and explicit locking. The functions for selecting records are not currently included in the framework, but are available in the sample programs and work with the framework. All the code for creating new records, modifying, saving, and deleting them, plus numerous utility functions and the implementation of the appwindow data type are also part of the library. The appwindow type is also suitable for building programs that have either only one main window, or applications with multiple top-level windows.

Some of the more useful utility functions, examples of which can be seen in the Address Book sample program, are:

- `checkneedsave()`
- `getappwindowfromwindow()`
- `getmenuitemwindow()`
- `defer()`

- `findfirstfocusablecontrol()`
- `lookup()`
- `gettableformatstrings()`

This next list contains the functions that are used specifically as part of the data-entry process and which are meant to be directly associated with menu and tool bar items:

- `newrecord()`
- `duplicaterecord()`
- `modifyrecord()`
- `saverecord()`
- `deleterecord()`

The two main data types that are provided by the application framework are the appwindow and the application. The application type is meant to provide a container for all of the things that might be needed throughout the application. This includes locale information, default format strings for data type conversion, the operating system type, a ring of data sources, information about the system (display size, named system color values, etc.), a placeholder for the window icon bitmap, and the title of the application. This type is meant to be incorporated in a user program's application type, if the needs of the user program exceed what is provided in the standard type.

The appwindow type is meant to contain information specific to the window. That includes the tables that are opened as part of it. It also contains a reference to the application object (or your own application object). It stores some state information, such as the current table, the last selected value according to the current index, the last value of the internal unique key for identifying records, whether the window is currently in fast selection mode (either fast forward or rewind), a reference to the physical window that it wraps, as well as references to the menu bar, tool bar, and status bar objects, for fast access. It also includes a number of very useful methods, such as `findtable()`, `closeall()`, `opendatatable()`, and `openformdirect()`.

For full implementation details of these two types, as well as the others that are included, examine the source code to the application framework library, which is provided as part of the distribution and see the section called “application” and the section called “appwindow”.

Warning

Do not change the source code to the appframework or the other libraries unless absolutely necessary! They have been carefully designed to work correctly and to permit extensions to be added using the event mechanism. If you need to make a change to a library to fix a bug, it is best to report it to Superbase Software Limited via email or on the forum and have us assess the problem. If you just want use a different function, then add it to the application or create a library of your own.

In summary, a typical application framework program will initialize the application object, create an initial appwindow object, and as part of that produce a menu, tool bar, and status bar. At that point, the program will call the application.`run()` method and it will remain there until the application is closed down. While in the `run()` method, it will respond to events, such as menu and tool bar events, or form button events. These events may result in new forms being loaded into one standard window, or it may result in multiple windows being opened, each potentially with its own menu and tool bar. The design is up to you.