Chapter Two

SLX AS A PROCEDURAL LANGUAGE

2.1 Introduction

This chapter describes the SLX language as a procedural programming language. SLX's simulation features are introduced in Chapter 3.

C and to a lesser extent C++ influenced the development of SLX. For many programming constructs, SLX uses syntax identical to that of C/C++. Hence, readers familiar with C/C++ will be able to read much of this chapter very quickly. However, note that SLX *does* depart from C/C++ syntax in some cases, and SLX's support of some of their constructs is restricted. For a few C-like constructs, SLX's implementation is more general than C's.

Since SLX is much closer to C than C++, throughout the remainder of this book, when we compare SLX to C, we mean the C subset of C++. We'll only mention C++ when comparing SLX to C++ features not found in C. This will allow us to avoid the cumbersome "C/C++" notation.

Finally, note that what are called *functions* in C or *subroutines* in other languages are called *procedures* in SLX.

2.2 SLX Program Structure

An SLX program is a collection of one or more SLX *files*. An SLX file is a collection of one or more *modules*. A module is a collection of definitions of *global variables*, *classes*, *procedures*, and *macros*. Exactly one module must contain a definition of procedure **main**().

The following example illustrates a simple (single file, single module, single procedure) SLX program containing no user-defined variables:

```
module SimpleModule
    {
    procedure main()
        {
        print "Hello SLX World!\n";
        }
    }
```

2.2.1 SLX Source Files

An SLX program is a collection of files. If an SLX program comprises more than one file, the files are organized as a tree. Starting with a root file, each file may **import** additional SLX files. C users should note that SLX's **import** statement only superficially resembles C's **#include** statement, which simply incorporates into a file a verbatim copy of another file.

The following rules apply to the use of **import** statements:

- Any **import** statements in a file must be placed at the very beginning of the file, before any non-comment statements, i.e., before the first **module** in the file.
- **import** statements are processed in depth-first order. If a root file imports file A, and file A imports file B, and file B imports files C and D, the order in which files are processed is C, D, B, A. This approach assures that lower-level definitions are processed before higher-level code that depends on them.
- SLX automatically handles multiple imports of the same file. Once a file has been imported, any additional imports of the same file are ignored. (C/C++ users will find this a great relief from the sometimes onerous burden of suppressing multiple C/C++ **#include** statements.)
- SLX's development environment does not use a list of files comprising a "project." To compile an SLX program, you simply direct the compiler to compile a root file, and any files specified in **import** statements are processed automatically.
- By default, SLX's development environment compiles a program starting with the file in the active source window. If no source window is active, the compiler will complain.
- If you either purposely or inadvertently initiate compilation of a file that is not the root file of a program, you have to be careful about implied dependencies. For example, assume that a root file imports file A and file B. Further assume that neither file A nor file B contain any further imports. If you compile the root file, the order in which files are processed is A, B, root. If you attempt to compile file B directly, and file B requires definitions contained in file A, errors will occur, because B doe not import A. In other words, file B has an implied dependency on file A that is exposed only when you try to compile file B only. To deal with such problems, SLX provides an "Always Compile Root" option in its Options menu. The SLX compiler is extremely fast, so the time saved by compiling only part of a program is very small.
- The **import** statement takes two forms:

  ```
  import <systemfile.slx>
  import "myfile.slx"
  ```

  The first form is used for importing "system" files, including standard files provided with SLX. By default, system files are imported from the SLX directory. You can change this by setting the SLX environment variable to point to a different directory. The second form is used for importing user files.

  For both cases, the ".slx" file extension is optional.

2.2.2 SLX Modules

SLX modules contain definitions of global variables, classes, procedures, and macros. The primary reason SLX includes modules is to control access to the contents of the modules from other parts of an SLX program. For example, if you're developing a package for use by others, you might want to carefully choose which parts of your package are exposed to its users and which parts are hidden. You might choose to make certain variables read-only outside the module, allowing other parts of a program to read, but not write them.

Access to the contents of a module is controlled by providing prefixes. Any prefixes provided can be overridden on a case-by-case basis for items within a module. For example, in a read-only module, individual variables could be declared as writable. Available module prefixes are shown in Table 2-1:

Prefix	Meaning
public	Components of a module are visible outside the module.
private	Components of the module are invisible outside the module. ***Unless explicitly declared public, all modules are assumed to be private.***
read_only	If visible, variables can be read, but not written outside the module.
write_only	If visible, variables can be written, but not read outside the module.
read_write	If visible, variables can be both read and written outside the module.
precursor	When the SLX compiler encounters a precursor module, it suspends what it was doing, compiles the precursor module in its entirety, and resumes compilation of the program. Variables, procedures, macros, etc., defined in a precursor module are subsequently available for use during the compilation of the remainder of the program. Precursor modules are an advanced topic covered in Chapter 7.

Table 2-1 – Module Prefixes

The following example illustrates representative uses of module prefixes:

```
public read_only module Module1
    {
    procedure PublicProcedure()
        {
        …
        }

    private procedure PublicProcedure()           // usable only within Module1
        {
        …
        }

    read_write int     MyCounter;                 // read-write outside Module1
    }
```

2.2.3 Statements, Comments and Names

Each SLX statement must be terminated with a semicolon (";"). A sequence of statements enclosed between braces ("{ }") is called a compound statement. In any context that requires a statement, a compound statement may also be used. There are a handful of SLX constructs in which compound statements are required, not optional.

Comments are enclosed between "/*' and "*/". Everything between the starting and ending delimiters is ignored by SLX. Alternatively, "//" can be used as a comment delimiter, and all text from the "//" to the end of the current line is ignored by SLX.

SLX names are formed according to the following rules:

- The first character of a name must be alphabetic.
- Additional characters can be any combination of letters, digits or underscore ("_") characters.
- Names are case-sensitive; i.e., "MyCount" is different from "myCount".

2.2.4 Implications of Block Structure

SLX is a block-structured language. In many contexts, multiple uses of the same name with differing semantics are legal. When multiple definitions exist for a given name, the definition in the smallest enclosing context (scope) is used. If no enclosing context exists, or if you wish to override the enclosing context that would otherwise be chosen, you may have to supply a qualified name that uses the "::" operator. Example 2-3 shows the use of qualified names.

In the following example, the first two assignments are unambiguous, because the variable names are unique. The third assignment statement is unambiguous, because the scope of myProcedure more closely encloses the statement than the scope of Module3. In the fourth assignment statement, the default selection of the narrowest containing scope is overridden. In the last two assignment statements, the use of a qualified name is mandatory, since neither Module1 nor Module2 enclose the ambiguous reference to myCounter1.

```
public module Module1
    {
    int      myCounter1,
             myCounter1A;
    }

public module Module2
    {
    int      myCounter1,
             myCounter1B;
    }

public module Module3
    {
    int      myCounter2;

    procedure myProcedure()
        {
        int      Count, myCounter2;
        …
        Count = MyCounter1A;            // Unique to Module1
        Count = MyCounter1B;            // Unique to Module2
        Count = MyCounter2;             // local to the current procedure
        Count = Module3::MyCounter2;    // myCounter2 from Module3
        Count = Module1::MyCounter1;    // myCounter1 from Module1
        Count = Module2::MyCounter1;    // myCounter1 from Module2
        }
    }
```

2.3 Simple Data Types

This section describes SLX's five basic data types. For each simple data type, the following information is presented:

- A definition of the type
- User-defined and system-defined constants for the type
- Sample declarations of variables of the type
- Expression-level operators that apply specifically to the type
- Statements that apply specifically to the type

The following general rules apply to all data types:

- All variables have a defined initial value. This is a departure from C and many other languages, in which uninitialized variables have random values.
- Variable declarations may include the optional prefixes shown in Table 2-2:

Prefix	Meaning
public	The variable is visible outside its containing scope.
private	The variable is invisible outside its containing scope.
read_only	If visible, the variable can be read, but not written outside its containing scope.
write_only	If visible, the variable can be written, but not read outside its module.
read_write	If visible, the variable can be both read and written outside the its containing scope.
constant	The variable is assigned a mandatory initial value and cannot be changed thereafter.
control	The variable is a state variable. State variables are used in conjunction with SLX's **wait until** statement, which will be presented in Chapter 4. At each point in a program where the value of a control variable is changed, SLX inserts a check to see whether any ongoing activity is presently waiting for the variable to attain a given value.
static	For variables used inside procedures or SLX classes, the "static" prefix indicates that a single copy of the variable is shared among all instances of the class or procedure. For non-static variables in classes or procedures, a separate instance of the variable is allocated for each instance of the class or procedure. Variables defined at the module level (outside all classes and procedures) are inherently static.
OEM	Variables with an OEM prefix are hidden from all SLX debugger displays. SLX uses OEM variables in many of its predefined classes to prevent users from snooping at details about which they needn't know.

Table 2-2 Variable Declaration Prefixes

Initial values for variables can be supplied in declarations, e.g.,

```
    int         i =  1;
    double      x = 100.0;
```

Initial values can be specified as expressions, subject to the limitation that any variables used in the expressions must be initialized in prior *statements*. "Earlier in the *same* statement" is not allowed. Consider the following examples:

```
    int         i= 1, j = 2;
    int         k = i + j;                  // k = 3
    double      x = 1.0, y = 2.0, z = x + y;   // compile-time error (limitation described above)
```

2.3.1 Integer Variables (ints)

SLX provides 32-bit, signed integer variables. SLX does not include C's 16-bit ("short int"), 8-bit ("char"), or 64-bit ("_int64") integers. There are two reasons SLX includes only 32-bit integers. First, rigorous run-time error checking would require checking all assignments of wider integer variables or expressions into narrower integer variables to assure that the data would fit in a narrower container. Most SLX programs have sufficient intentional randomness that if wider integer values were truncated without complaint to fit into narrower containers, the truncation errors would be extremely difficult to find. Run-time error checking would be expensive. Second, memory is cheap. The benefit of small memory savings is easily outweighed by the cost of increased execution time.

The maximum value of an integer variable is 2147483647, and the minimum value is -2147483648.

Integer constants can be specified as signed decimal, or unsigned hexadecimal or binary values. Hexadecimal constants are prefixed with "0x" or "0X" and are comprised of the digits 0…9 and letters A…F in either upper or lower case. Binary constants are prefixed with "0b" or "0B" and are comprised entirely of zeros and ones. The following are examples of integer constants:

```
1234
-99
0x10AB              // hexadecimal
0b100100111         // binary
```

The following are SLX's predefined integer constants:

Symbol	Meaning
INT_MIN	The minimum allowable integer (-2147483648)
INT_MAX	The maximum allowable integer (2147483647)

Table 2-3 Predefined Integer Constants

The following are sample declarations of integer variables:

```
int            i, j, k;
control int    myCounter;          // control variable
int            CountDown = 10;     // initial value given
constant int   NCASES = 100;       // constant (cannot be changed)
```

SLX's expression-level integer operators are shown in Table 2-4. Operators are shown in the order of their precedence; i.e., in the absence of parentheses, operators of higher precedence are processed before operators of lower precedence. Operators of equal precedence are processed in left-to-right order. For example, "-i+j*k" is evaluated as "(-i)+(j*k)", and "i–j+k" is evaluated as "(i-j)+k".

SLX's statement-level integer operators are shown in Table 2-5, and its integer built-in functions in Table 2-6. Operators for comparing integers are shown in Table 2-13.

Operator	Result (Assuming j = 5 and k = 2)	Prece-dence	Meaning
i = ++j;	i = 6, j = 6	7	j is preincremented by one
i = --j;	i = 4, j = 4	7	j is predecremented by one
i = j++;	i = 5, j = 6	7	j is postincremented by one.

Operator	Result (Assuming j = 5 and k = 2)	Prece-dence	Meaning
i = j--;	i = 5, j = 4	7	j is postdecremented by one
i = +j;	i = 5	6	Unary plus
i = -j;	i = -5	6	Unary minus (2's complement)
i = ~j;	i = 0xfffffffa	6	The inverted bits of j (logical not)
i = j * k;	i = 10	5	j times k
i = j / k;	i = 2	5	j divided by k (truncated)
i = j % k;	i = 1	5	The remainder after dividing j by k (modulus division). Note that the sign of the remainder is the same as the sign of the quotient, so –3 % 2 is –1 (-3 divided by 2 = -1, with a remainder of –1)
i = j + k;	i = 7	4	j plus k
i = j – k;	i = 3	4	j minus k
i = j << k;	i = 20	3	j shifted k bits to the left
i = j >> k;	i = 1	3	j shifted k bits to the right
i = j & k;	i = 0	2	The bits of j AND'ed with the bits of k
i = j | k;	i = 7	1	The bits of j OR'ed with the bits of k
i = j ^ k;	i = 7	1	The bits of j EXCLUSIVE OR'ed with the bits of k

Table 2-4 – Expression-Level Integer Operators

Operator	Result (Assuming i = 5 and j = 2)	Meaning
i = j;	i = 2	simple assignment
i += j;	i = 7	j is added to i
i -= j;	i = 3	j is subtracted from i
i *= j;	i = 10	i is multiplied by j.
i /= j;	i = 2	i is divided by j.
i %= j;	i = 1	i is assigned the remainder of i divided by j (modulus division).
i |= j;	i = 7	i is assigned the bits of i OR'ed with the bits of j.
i &= j;	i = 0	i is assigned the bits of i AND'ed with the bits of j.
i ^= j;	i = 7	i is assigned the bits of i EXCLUSIVE OR'ed with the bits of j.
i <<= j;	i = 20	The bits of i are shifted left j bits.
i >>= j;	i = 1	The bits of i are shifted right j bits.
++i;	i = 6	i is incremented by one.
i++;	i = 6	i is incremented by one. When used as free-standing statements, pre- and post-incrementing are equivalent.
--i;	i = 4	i is decremented by one.
i--;	i = 4	i is decremented by one. When used as free-standing statements, pre- and post-decrementing are equivalent.

Table 2-5 – Statement-Level Integer Operators

Function	Meaning
abs(i)	The absolute value of i
ascii(i)	Converts an integer i, where i is in the range of 0...255, into an ASCII string of length 1.
pow(i, j)	i raised to the j'th power

Table 2-6 – Integer Built-In Functions

2.3.2 Floating Point (Double) Variables

SLX provides 64-bit floating point variables. ***SLX does not include C's 32-bit ("float") floating point variables. Variables declared as* float *are treated as* double**. SLX excludes 32-bit floating point variables because they offer inadequate precision for discrete event simulation. Consider a year-long simulation of a factory in which the time unit is one second. One year = 365 x 24 x 60 x 60 = 31,536,000 seconds. 32-bit floating point arithmetic offers about 6-7 decimal digits of precision. A year's worth of seconds requires 8 digits, so after a year's worth of simulation, we can keep track of time to roughly the nearest 10 seconds. Any objects that are moving over time require additional digits of precision to keep track of their positions. With 32-bit floating point arithmetic, this rapidly becomes impossible in many simulations.

The maximum magnitude of a floating point variable is approximately 1.797E+308, and the minimum magnitude is approximately 1.797E-308.

The following are examples of floating point constants:

```
48000.0
-2.5
1.23E3          // 1230.0
1e-3            // .001
```

The following are SLX's predefined floating point constants:

Symbol	Meaning
DBL_MIN	The minimum allowable value (-1.797E+308)
DBL_MAX	The maximum allowable value (1.797E+308)
INFINITY	A hardware-defined constant greater than all other values
PI	3.14159…

Table 2-7 Predefined Floating Point Constants

The following are sample declarations of floating point variables:

```
double              x, y, z;
float               xmax;                       // treated as double
control double      RemainingCapacity;          // control variable
write_only double   CountDown = 10;             // initial value given
constant double     ShutdownTime = 48000.0;     // (cannot be changed)
```

SLX's expression-level floating point operators are shown in Table 2-8. Operators are shown in the order of their precedence; i.e., in the absence of parentheses, operators of higher precedence are processed before operators of lower precedence. Operators of equal precedence are processed in left-to-right order. For example, "-x+y*z" is evaluated as "(-x)+(y*z)", and "x–y+z" is evaluated as "(x-y)+z".

SLX's statement-level floating point operators are shown in in Table 2-9, and its floating point built-in functions in Table 2-10. Note that all trigonometric functions use radian measure. Operators for comparing floating point values are shown in Table 2-13.

Operator	Result (Assuming y = 5.5, z = 2.0, and i = 2)	Prece-dence	Meaning
x = ++y;	x = 6.5, y = 6.5	4	y is preincremented by 1.0
x = --y;	x = 4.5, y = 4.5	4	y is predecremented by 1.0
x = y++;	x = 5.5, y = 6.5	4	y is postincremented by 1.0.
x = y--;	x = 5.5, y =- 4.5	4	y is postdecremented by 1.0
x = +y;	x = 5.5	3	Unary plus
x = -y;	x = -5.5	3	Unary minus
x = y * z;	x = 11.0	2	y times z
x = y / z;	x = 2.75	2	y divided by z
x = y % z;	x = 1.5	2	The remainder after dividing y by z and truncating the quotient to an integer (modulus division). Note that the sign of the remainder is the same as the sign of the dividend, so –3.2 % 2.0 is –1.2 (-3.2 divided by 2.0 truncates to -1.0, yielding a remainder of –1.2)
x = y + z;	x = 7.5	1	y plus z
x = y – z;	x = 3.5	1	y minus z
x = **pow**(y, i);	x = 30.25	0	Floating point y raised to the integer j'th power
x = **pow**(i, y);	x = 45.2548	0	Integer i raised to the floating point y'th power
x = **pow**(y, z);	x = 30.25	0	Floating point y raised to the floating point z'th power

Table 2-8 – Expression-Level Floating Point Operators

Operator	Result (Assuming x = 5.5 and y = 2.0)	Meaning
x = y;	x = 2.0	simple assignment
x += y;	x = 7.5	y is added to x
x -= y;	x = 3.5	y is subtracted from x
x *= y;	x = 11.0	x is multiplied by y.
x /= y;	x = 2.75	x is divided by y.

Operator	Result (Assuming x = 5.5 and y = 2.0)	Meaning
x %= y;	x = 1.5	x is assigned the remainder of x divided by y (using truncated division).
++x;	x = 6.5	x is incremented by 1.0.
x++;	x = 6.5	x is incremented by 1.0. When used as free-standing statements, pre- and post-incrementing are equivalent.
--x;	x = 4.5	x is decremented by 1.0.
x--;	x = 4.5	x is decremented by 1.0. When used as free-standing statements, pre- and post-decrementing are equivalent.

Table 2-9 – Statement-Level Floating Point Operators

Function	Meaning
abs(x)	Absolute value of x
acos(x)	Arc Cosine of x
asin(x)	Arc Sine of x
atan(x)	Arc Tangent of x
atan2(y, x)	Arc Tangent of y / x (x can be zero.)
ceil(x)	The next integer greater than or equal to x (for positive x) The next integer less than or equal to x (for negative x)
cos(x)	Cosine of x
floor(x)	The next integer less than or equal to x (for positive x) The next integer greater than or equal to x (for negative x)
log(x)	Natural (base e) logarithm of x
log10(x)	Logarithm (base 10) of x
pow(x, i)	Floating point x raised to the integer j'th power
pow(i, y)	Integer i raised to the floating point y'th power
pow(x, y)	Floating point x raised to the floating point y'th power
round(x)	Floating point x rounded to the nearest integer
sin(x)	Sine of x
sqrt(x)	Square Root of x
tan(x)	Tangent of x

Table 2-10 – Built-In Math/Trig Functions

2.3.3 Character String Variables

Unlike C, in which strings are treated as arrays of characters, SLX provides built-in string operations for copying, concatenating, and translating strings; extracting substrings; and converting integers to strings.

Every string has a declared maximum length and a *current* length. The maximum length of a character string is 32767 characters. Assignments into string variables that exceed the string variable's maximum length are truncated without complaint. String variables for which an initial value is supplied can specify a length of "*", meaning that SLX will set the maximum length of the string to that of its initial value. String variables that are used as formal parameters of SLX procedures *must* be defined with a maximum length of "*". (The maximum length of a string parameter is taken from the string variable that is passed as an argument. In other words, you cannot override the maximum length of a passed string.)

By default, string variables are initialized as empty; i.e., their current lengths are set to zero.

2.3.3.1 String Constants

String constants start and end with quotation marks ("double" quotes). Within a string, the "escape sequences" shown in Table 2-8 are allowed. Each escape sequence starts with a backslash ("\") character. ***If SLX encounters consecutive string constants, it combines them into a single string constant, inserting a newline ("\n") at the boundaries between consecutive strings***. This is extremely convenient for use with SLX picture output (See Section 2.9.2.1 Picture Format Processing); however ***it is incompatible with C, which combines consecutive strings with no such insertions.***

The following are examples of string constants:

```
"TOM"
"Output string ending with a newline\n"
"Output string containing a \Rred\R word"
"Output string containing a \"quoted\" word"

"Line 1" "Line 2\n" equivalent to   "Line1\nLine2\n"
```

2.3.3.2 Sample Declarations of String Variables

```
string(10)      Mystring;
string(20)      Format = "The value of x is _.__\n";
string(*)       Text = "SLX calculates the length";
```

Escape Sequence	Notes	Meaning
\"		Allows inclusion of a quotation mark in a string constant. (Without the "\", a string constant would end.
\\		Allows inclusion of a backslash character in a string constant.
\a	1	An ASCII "bell" character
\b	1	An ASCII backspace character
\f	2	An ASCII formfeed character
\n		A newline character (used to terminate a line of output)
\r	1	An ASCII carriage return character
\t	3	An ASCII "tab" character
\v	2	An ASCII vertical tab character
\xhh	4	The hexadecimal value "hh" is inserted.
\B	5	Starts/ends text to be written in boldface
\I	5	Starts/ends text to be written in italics
\R	5	Starts/ends text to be written in a red color

Escape Sequence	Notes	Meaning
\U	5	Starts/ends text to be written with underscores
_	6	Allows inclusion of an underscore ("_") in picture format output
*	6	Allows inclusion of an asterisk ("*") in picture format output
\#	7	Allows inclusion of a pound sign ("#") in macro-generated text

Notes	Details
1	These characters are probably obsolete. They have unpredictable effects when written to the screen or a file, and may or may not work when written to a printer.
2	Preserved if written to a file; meaningful only if written to a printer.
3	Tab characters should be used only for constructing output files to be cut and pasted into documents that contain meaningful tab stops. You should not use tab characters in SLX picture format output.
4	Use with caution!
5	These escape sequences are meaningful only if written to the screen or written to an RTF output file. (See details in Section 2.9.2.4 RTF output)
6	SLX uses "_" and "*" characters to designate picture format output fields into which values are to be edited. If you need to place literal underscore or asterisk characters into a picture and not have them interpreted as editing fields, you must use these escape sequences. (See details in Section 2.9.2.1 Picture Format Processing).
7	SLX uses "#" to designate macro output fields into which values are to be edited. If you need to place literal "#" characters in macro-generated text and not have them interpreted as editing fields, you must use this escape sequence. (See details in Section7.?)

Table 2-11 – String Constant Escape Sequences

2.3.3.3 Expression-Level String Operators

2.3.3.3.1 Concatenation

The **cat** operator concatenates two strings:

```
string(5)     x = "abcde",
              y = "fghij";
string(8)     z;

z =  x cat y ;  // Result: z = "abcdefgh"     (Note truncation to 8 characters)
```

2.3.3.3.2 Substring Extraction

The **substring** operator is used to extract a substring of a string. There are two forms of the **substring** operator:

substring(*source_string, starting_position, length*)

and

substring(*source_string, starting_position*)

Each form specifies the source string from which a substring is to be extracted and the position (1…N) at which the substring begins. In the first form, an explicit length is supplied, and in the second form, the implied length is that of the remainder of the string. In the first form, the specified length must be greater than or equal to zero, and the combination of starting position and length cannot extend beyond the maximum length of the source string. The combination of starting position and length *can* extend beyond the current length of the string, in which case the substring is padded with blank characters.

The following are examples of the **substring** operator:

```
string(10)     x = "123456789";
string(10)     z;

z = substring(x, 2, 4);       // Result : "2345"
z = substring(x, 7, 4);       // Result : "789 "            (Note: past end => blank added)
z = substring(x, 3);          // Result : "3456789"
```

2.3.3.4 Statement-Level String Operators

2.3.3.4.1 Concatenation Assignment

The **cat**= operator is used to append a string onto another:

```
string(15) x = "Walla ";

x cat= "Walla";             // Result : x = "Walla Walla"
x cat= ", WA";              // Result : x = "Walla Walla, WA"
```

2.3.3.4.2 Substring Assignment

The **substring** operator can be used to specify that a string assignment is to be into a substring of a string variable. Two forms of the **substring** operator can be used for substring assignment:

substring(*target_string, starting_position, length*) = *source_string*;

and

substring(*target_string, starting_position*) = *source_string*;

The restrictions on starting position and length are identical to those of the expression-level **substring** operator.

The following are examples of **substring** assignment:

```
string(10) x = "123456789";
string(10) y = "ABCD";

substring(x, 2, 3) = y ;      // Result : x = "1ABC56789"
substring(x, 3, 6) = y ;      // Result : x = "12ABCD  9"   (y extended with 2 blanks)
```

2.3.3.5 Built-In String Functions

2.3.3.5.1 Determining the Current and Maximum Lengths of a String

The length built-in function returns the current length of a string, and the str_maxlen built-in function returns the maximum length of a string. When applied to a string constant or string expression, str_maxlen returns a value identical to the length function.

The following are examples of length and str_maxlen:

```
string(10) x = "123456789";
string(10) y = "ABCD";

j = length(x);              // Result: j = 9
j = str_maxlen(x);          // Result: j = 10
j = length(y cat y);        // Result: j = 8
```

2.3.3.5.2 Creating Special Characters by Number

The ascii built-in function converts an integer character number into a string of length 1. Note that the ascii and str_ivalue built-in functions are inverses.

```
string(1)   x;

x = ascii(0x41);     // Result: x = "A"
```

2.3.3.5.3 Converting Character Strings to Integers

The str_ivalue built-in function returns the integer character number of a specified ascii character. The character must be a string of length 1. Note that the str_ivalue and ascii built-in functions are inverses.

```
int       j;

j = str_ivalue("A");              // Result: j = 0x41
```

2.3.3.5.4 More General Forms of Conversion to and from Strings

The **read string**= and **write string**= statements can be used to accomplish a wider variety of conversions of SLX data to and from strings. (See Sections 2.9.2.7 String Output and 2.9.3.2 Reading from Strings)

2.3.3.5.5 Character Translation

The str_translate built-in function is used to translate the characters comprising a string in accordance with a translation object that specifies a collection of prototype characters and the characters to which characters comprising the prototype are to be translated. You can think of the translation as being driven by a 2-column table. During the translation operation on a character string, every single character is compared to the *prototype* column. If a source character matches a character in the prototype column, the source character is replaced by the character contained in the same row of the *substitution* column.

The following example illustrates the use of str_translate:

```
module basic
    {
    // Definition of translation table

    translation     UpperCase("aeiou", "AEIOU");

    string(300)     test_string = "The act or process of translating";          // Source string

    procedure main
        {
        print (test_string) "_\n";
        str_translate(test_string, UpperCase);                                  // Translation
        print (test_string) "_\n";
        }
    }
```

The output of this program is:

```
The act or process of translating
ThE Act Or prOcEss Of trAnslAtIng
```

2.3.3.5.6 Searching for Text within Strings

The str_pos built-in function is used to search for strings inside other strings:

```
string(10) Source = "ABCDEFGHIJ";
int           i;
i = str_pos(Source, "EFG");         // Result: i = 5
```

The str_pos function returns the position (1…N) of a search string within another string. If the search string is not found str_pos returns a value of zero.

2.3.3.5.7 String Hash Functions

Hash functions are used to map strings into integers. Hash functions are typically used to shorten searches when looking up keywords. The hash value of a string is used as a starting index into a table that contains for each index value a list of keywords that hash into that index. Assume that you have a list of 100 keywords, each of which is equally likely to be the target of a keyword search. If you simply search the list of keywords in order each time you need to look up a keyword, the average search will be of length 50. On the other hand, if you use a hash function that maps your keywords into values in the range of 1…100, the average search length will be considerably shorter. Exactly how much shorter depends on the quality of the hash function. In the case we're discussing, a perfect hash function would achieve unique mappings for all 100 keywords, and the average search length would be 1. Perfect hash functions are, of course, impossible to achieve. SLX contains a good, general-purpose hash function:

```
int      i;
i = str_hash("MyKeyword", 100);      // Result: i = a value In the range of 1…100
```

2.3.4 Enumerated Types

If you've never encountered a language that includes enumerated types, you should read this section *very* carefully, since enumerated types are extremely useful in constructing simulation models. If you know C, you should also read this section carefully, inasmuch as SLX's enumerated types go beyond those of C.

An enumerated type is simply a list of names that collectively comprise a type. An enumerated type can be specified in two ways, one intended for repeated use and the other for single use. The following is an example of the repeated use form, which is the more frequently used form:

```
typedef enum { IDLE, RUNNING, BROKEN } MachineState;

MachineState  LatheState, DrillpressState;

LatheState = IDLE;
```

The example above defines a type named MachineState that comprises three values: IDLE, RUNNING, and BROKEN. SLX initializes enumerated type variables with a system-defined value of **NONE**. The variables LatheState and DrillpressState can be assigned the values IDLE, RUNNING, BROKEN, or **NONE**, and *only* these values. In other words, all the possible values are *enumerated* in the type definition.

The second form of enumerated type definition is designed for single use:

```
enum { RED, GREEN, BLUE }    Color;
```

In the above example, Color is a variable, not a type. The variable Color can be assigned the values RED, GREEN, BLUE, or **NONE**, and *only* these values.

Note that the same type values can be used in more than one enumerated type:

```
typedef enum { RED, GREEN, BLUE }            MainColor;
typedef enum { PURPLE, RED, BLACK }          SecondaryColor;

MainColor         color1;
SecondaryColor    color2;

Color1 = RED;
Color2 = RED;        // not the same RED as above
Color1 = BLACK;      // illegal
Color2 = GREEN;      // illegal
```

In languages that lack enumerated types, states of the type shown above are usually represented as integers:

```
int  LatheState;           // State variable is an integer

LatheState = 1;            // IDLE
LatheState = 2;            // RUNNING;
LatheState = 79;           // Legal integer value that has no known meaning
```

Some languages allow specification of symbolic constants (named integers). In SLX, this can be accomplished by defining integer constants:

```
constant int    IDLE = 1, RUNNING = 2, BROKEN = 3;        // Integer constants
constant int    RED = 1, GREEN = 2, BLUE = 3;             // Integer constants
int             LatheState;                               // Integer variable

LatheState = IDLE;         // OK
LatheState = 27;           // Valid integer assignment, but meaningless
LatheState = BLUE          // Valid integer assignment; equivalent to LatheState = BROKEN;
```

Because all the named values in the above example are integers, it's extremely easy to assign values that are meaningless or very misleading to integer variables. ***You simply* must *learn to use enumerated types***, for the following reasons:

- They *greatly* improve program readability.
- They offer air-tight compile-time error checking. (You cannot assign a Color to a MachineState variable, and you cannot assign an unnamed integer to any enumerated type variable.
- Enumerated types can be used to dimension arrays. A simple example is shown below. See Section 2.6.2 Array Declarations for details.

```
double          HoursInState[MachineState];
```

The above example defines a one-dimensional floating point array that is indexed by MachineState variables or constants. You cannot use integer subscripts with this array. Furthermore, suppose you decide to add additional states to the MachineState enumerated type, e.g., UNDERGOING_MAINTENANCE. If you simply add the additional state to the enumerated type, the HoursInState array will be automatically resized to reflect the additional allowable index.

2.3.4.1 Enumerated Type Operators

```
typedef enum { RED, GREEN, BLUE }              MainColor;
typedef enum { PURPLE, RED, BLACK }            SecondaryColor;
typedef enum { IDLE, RUNNING, BROKEN }         MachineState;
```

Operator	Meaning	Example	Value
first *EnumeratedType*	The first value defined for an enumerated type	**first** MainColor	**RED**
last *EnumeratedType*	The last value defined for an enumerated type	**last** MachineState	**BROKEN**
successor(*EnumerationValue***)**	The value after a given value	**successor(**IDLE**)**	**RUNNING**
		successor(BROKEN**)**	**NONE**
successor(*EnumerationValue***)** **in** *EnumeratedType*	Same as above, but enumerated type is required to remove ambiguity	**successor(**RED**) in** MainColor	**GREEN**
		successor(RED**) in** SecondaryColor	**BLACK**
predecessor(*EnumerationValue***)**	The value after a given value	**predecessor(**RUNNING**)**	**IDLE**
		predecessor(IDLE**)**	**NONE**
predecessor(*EnumerationValue***)** **in** *EnumeratedType*	Same as above, but enumerated type is required to remove ambiguity	**predecessor(**GREEN**)** **in** MainColor	**RED**
		predecessor(GREEN**)** **in** SecondaryColor	**Compile-Time Error!**

Table 2-12 – Enumerated Type Operators

2.3.4.2 Augmenting a Previously Defined Enumerated Type

Subsequent to defining an enumerated type, you can add additional type values by using the following syntax:

typedef enum { *new_constant,…* } **augment** *old_type_name*;

e.g.,

typedef enum { BEING_SERVICED } **augment** MachineState;

2.3.4.3 Enumerated Type Statements

The only statement designed exclusively for use with enumerated types is the enumerated type form of the **for each** statement, which takes the following form:

```
for (EnumerationVariable = each [ reverse ] EnumerationType
    [ { before | after | from } enum_expression ]
    [ with Boolean_expression ] )
        statement
```

- The optional **from** clause specifies the starting point of the iteration.
- The optional **reverse** keyword causes iteration to take place in reverse order, i.e., from the last enumerated type (or the **from** point) to the first enumerated type.

- The optional **before/after** specify the starting point of the iteration as **before/after** a specified value of the enumerated type. Note that the sense of **before/after** is not affected by the **reverse** keyword; i.e., **before/after** clauses are interpreted in the usual sense *prior* to starting a **reverse** iteration.
- If you use **before**, you must also use **reverse**.
- If you use **after**, you cannot use **reverse**.

The following are examples of the enumerated type version of the for each statement:

```
MachineState  m;

for (m = each MachineState after IDLE)
    print (HoursInState[m], m)     "__._ hours spent in state _\n";

for (m = each MachineState with m != IDLE)
    print (HoursInState[m], m)     "__._ hours spent in non-IDLE state _\n";
```

2.3.5 Boolean Variables, Expressions, and Comparisons

Booleans variables and expresssions have **TRUE/FALSE** values. **TRUE** and **FALSE** built-in SLX constants. Boolean variables are initialized to **FALSE**. The following are sample declarations of Boolean variables and assignments to them:

```
boolean             MachineIdle = TRUE;          // initial value given
control boolean     ServerBusy;                  // used in wait until statements

MachineIdle = FALSE;
ServerBusy = TRUE;
```

2.3.5.1 Comparisons

SLX provides the obvious six comparison operators:

Operator	Meaning
a < b	**TRUE** if a is less than b
a <= b	**TRUE** if a is less than or equal to b
a == b	**TRUE** if a is equal to b
a != b	**TRUE** if a is not equal to b
a >= b	**TRUE** if a is greater than or equal to b
a > b	**TRUE** if a is greater than b

Table 2-13 – Comparison Operators

The following rules apply to comparisons:

- Comparison of integer and floating point values is allowed. An integer compared to a floating point value is converted to floating point.

- For all other comparisons, a and b must be of the same type.
- Special considerations apply to comparisons of variables with **time**, SLX's simulator clock, in **wait until** statements. These considerations are discussed in Section 4.?
- While all six forms of comparison are allowed for pointers, it is highly unlikely that comparisons other than equal or not equal are useful.
- When two character strings of unequal length are compared, an initial comparison is made using the shorter of the two lengths. If the strings are equal through the shorter length, the longer string is considered greater than the shorter.

2.3.5.2 Boolean Expressions

The terms of Boolean expressions can be comparisons, Boolean variables, or Boolean constants. Terms can be combined using and, or, and not operators. Operator precedence applies. For example, "!a && b || c" is evaluated as "((!a) && b) || c".

Operator	Result (Assuming a is TRUE and b is FALSE	Prece-dence	Meaning
a || b	**TRUE**	1	**TRUE** if either a ***or*** b is **TRUE**
a && b	**FALSE**	2	**TRUE** only if a ***and*** b are both **TRUE**
!a	**FALSE**	3	**TRUE** if a is ***not*** **TRUE**

Table 2-14 – Boolean Operators

SLX allows the use of keywords **and**, **or**, and **not** in place of the C-style "&&", "||", and "!" operators, e.g., "**not** a **and** b **or** c".

The following are examples of the use of Boolean expressions:

```
int             i;
double          x;
string(10)      PartName;
boolean         b;

if (i > 10 && x <= 32.5)
    …

if (b)
    ...

b = x != 25.0;              // assignment to a boolean variable
if (PartName == "Bolt")     // string comparison
    ...
```

2.4 Classes, Objects, and Pointers

A *class* serves five purposes:

- A class is a prototype for creating a specific kind of *object*. An object, in turn, is an *instance* of a class.
- A class contains attributes that collectively describe the state of each instance of the class. Classes provide a rich framework for describing components of a system.
- A class can contain attributes that are common to (shared by) all objects of the class.
- A class can contain user-defined *methods*, which are procedures that are called to perform actions on an object of the class.
- A class can contain *properties*, which are SLX standard methods. Some properties are automatically invoked by SLX when certain operations are performed on objects. Other properties are invoked only upon user request.

We should always be careful to distinguish the difference between an object, which is an instance of a class, and the class itself, which is a prototype for the object. However, when we speak informally, we sometimes blur this important distinction.

SLX makes very heavy use of classes and objects. Indeed, SLX is object-*based*. However, SLX is not an object-*oriented* language. It lacks the following capabilities that are characteristic of truly object-oriented languages:

- SLX does not support *inheritance*. Inheritance is a paradigm in which more specific objects inherit properties of more general objects. For example, in the inheritance paradigm, car, truck and bus classes might all be implemented as specializations of a vehicle class. SLX uses a *composition* paradigm, in which larger classes include instances of smaller classes. In SLX, one might define a class that describes a vehicle's route and then incorporate an instance of the route class into a class representing a bus. In the inheritance paradigm, exactly the opposite approach would be used. A route would be a property of a vehicle, and a bus would be a kind of vehicle.
- SLX does not support *polymorphism* (at least not in the classic sense). A sorting algorithm is an example of something that could be implemented in polymorphic fashion. Using a polymorphic approach, a single description of how "things" are to be sorted could be applied to integer, floating point, and/or character string data. In other words, the algorithm is the same, independent of the type of data on which the algorithm operates.

2.4.1 Definition of classes

A class definition takes the following form:

```
[ optional_prefixes ] class class_name  [ ( optional_parameter_declarations ) ]
        {
        attribute_definitions…
        property_definitions…
        method_definitions…
        }
```

Attributes, properties, and methods of a class can be specified in any order, except that attributes referenced in properties and methods must be defined before being referenced.

A sample class definition is shown below:

```
class widget (int initial_i)
    {
    static int  TotalWidgets;     // static attribute (shared by all widgets)
    int         i;                // local attribute (one for each widget)

    initial                       // invoked when the object is created
        {
        i = initial_i;            // copy argument to local attribute
        ++TotalWidgets;           // keep a running total of the number of widgets
        }

    report                        // invoked by the "report" statement
        {
        print (TotalWidgets)      "A total of _ widgets were created\n";
        }
    }
```

The prefixes shown in Table 2-15 can be used in class definitions:

Prefix	Meaning
active	The class contains or can be augmented (See Section 2.4.9 Augmenting the Definition of a Previously Defined Class) to contain an *actions* property. An actions property is executable code that describes an object's behavior when the object is *activated*. Actions properties and object activation are covered in detail in Chapter 3.
passive	The class does not contain an actions property, nor can it be augmented (See Section 2.4.9 Augmenting the Definition of a Previously Defined Class) to contain an actions property. Passive objects have no executable behavior of their own; they can only be acted upon by other objects.
public	Attributes and methods of the class are visible outside the module in which the class is defined.
private	Attributes and methods of the class are invisible outside the module in which the class is defined. In the absence of a **private** prefix, attributes are assumed to be **public**.
read_only	If visible, class attributes can be read, but not written outside the module in which the class is defined.
read_write	If visible, class attributes can be both read and written outside the module in which the class is defined.
write_only	If visible, class attributes can be written, but not read outside the module in which the class is defined

Table 2-15 Class Prefixes

Arguments to a class allow customization of individual objects when they are created. In other words, creation can be parameterized. Definition of parameterized classes is discussed below; however, at this point you should note that arguments passed to a class do not persist beyond the object creation process; i.e., they are not considered *attributes* of the class. Any information passed via arguments that must persist over the lifetime of an object must be saved in attributes of the created object.

Attributes of a class are variables that are declared in exactly the same manner as ordinary SLX variables.

SLX provides the class properties summarized in Table 2-16. An example illustrating all of the properties except the **actions** property is given in Section 2.4.8 Example of the Use of Properties.

Property	Usage
initial	The **initial** property is invoked when an object is created.
final	The **final** property is invoked as the last action in the destruction of an object.
destroy	The **destroy** property is invoked when an object is the subject of a **destroy** statement. Note that requests to destroy an object are postponed if there are outstanding uses of the object. The primary use of the **destroy** property is to remove any such references so the object *can* be destroyed.
actions	The **actions** property specifies the executable behavior pattern of an active object. The actions property is invoked after an object has been **activate**d ***and*** the object's turn to execute arises (subject to priorities and many other considerations).
report	The **report** property is used for printing information collected inside an object. The **report** property is invoked when a **report** statement either directly or indirectly specifies an object of the class.
clear	The **clear** property is used for clearing statistics collected in and for an object. The clear property is invoked when a **clear** statement either directly or indirectly specifies an object of the class.

Table 2-16 Class Properties

2.4.2 Pointers

SLX pointer variables are similar to, but considerably restricted versions of, pointers in other languages, such as C and C++:

- SLX pointers can only point to *objects*, not atomic data types. For example, in SLX you cannot have a pointer to an integer. If you need a pointer to an integer value, you must define an object containing an integer variable.
- SLX does not permit arithmetic operations on pointers.
- All SLX pointers are initialized with a **NULL** value. If you are going to pass SLX data to library functions of your own design written in C or C++, please not that ***SLX's* NULL *pointers have a non-zero value***. They point to a hidden "null object." The reasons for this are presented in Section 2.4.6 Use Counts.

- SLX's syntax for declaring pointers is wordier that that of C/C++.

The motivations behind these design decisions are as follows:

- The restrictions made in the design of SLX's pointers do not significantly hamper SLX's abilities to manage large collections of objects.
- SLX is not intended for use as a systems implementation tool. Hence, having variables that point directly to hardware- or operating system-dictated data formats is not a requirement.
- SLX's restrictions allow it to perform air-tight run-time error checking of all pointer references. For example, SLX traps situations in which a program attempts to reference an object using a **NULL** pointer.
- Arithmetic operations on pointers are error-prone, in large part because of implied scaling. In C and C++, incrementing a pointer by 1 increments the memory address to which the pointer points by the size of the data to which the pointer points, e.g., by 4 for a pointer to a 4-byte integer. Improperly accounting for scaling is a very common error among beginners and can be troublesome even for experts.
- Perhaps the most commonplace use of pointers to raw data and pointer arithmetic in C and C++ is in the manipulation of character strings. SLX provides a built-in string data type that in many respects is easier to use because it operates at a higher level.

SLX provides two kinds of pointers, those that can contain only addresses of a objects of a specific type, and those that can contain addresses of any object type. The latter are sometimes referred to as *universal* pointers. The following are examples of pointer declarations:

```
class widget …

pointer(widget)    w1,w2;          // can point only to "widget" objects
pointer(*)         u1;             // universal pointer (can point to any object)
```

SLX provides two built-in pointers, **ME** and **ACTIVE**. When used inside an object property or method, **ME** points to the current instance of the object. **ACTIVE** refers to the currently executing active object. (See Section 4.? for details).

The following example illustrates a typical use of **ME**:

```
class widget
    {
    initial
        {
        place ME into WidgetSet;    // every widget created goes in this set
        }
    …
    };
```

2.4.3 Object Creation & Initialization

2.4.3.1 Using the *new* Operator to Dynamically Create Objects

The **new** operator creates an object and returns a pointer to the object:

```
pointer(widget)    w;
...
w = new widget(27);        // create a widget, passing 27 as an argument
```

2.4.3.2 Creating Global (Module-Level) Objects

A global object is created by declaring a variable that is an instance of the class, as follows:

```
module SampleModule
    {
    class widget(int InitialValue)
        {
        ...
        }

    widget ModuleLevelWidget(25);        // global widget
    }
```

2.4.3.3 Creating Objects Local to a Procedure or Other Objects

A local object is created in exactly the same manner as a global object, except that the context is a procedure or an object, rather than a module:

```
module SampleModule
    {
    class XYZ       // class definition
        {
        double      x, y, z;        // (x,y,z) position
        }

    class PathSegment
        {
        XYZ         origin,         // three objects contained within each PathSegment
                    destination,
                    current_pos;
        double      speed;          // local to PathSegment
        }
    }
```

2.4.3.4 How Local and Global Objects are Actually Implemented

In looking at the example immediately above, one might guess that the storage for the origin, destination, and current_pos XYZ objects were located within and considered to be part of the storage for their containing PathSegment object. In reality, the containing object contains only *pointers* to the respective XYZ objects, which are allocated *independently*. While this approach is less efficient than allocating everything in contiguous memory, it offers two advantages. First it simplifies object allocation, since all objects are allocated in exactly the same manner. Second, it allows SLX to release the memory for the containing object and its contained objects independently. The use of pointers to contained objects may result in their having use counts that are higher than that of their containing objects. Thus, situations in which a containing object or procedure activation record can be destroyed, but one or more of its contained objects cannot are fairly commonplace.

2.4.4 Supplying Initial Values for Local or Global Objects

This section describes a way to supply initial values for an object. See also the discussion of the **initial** property in Section 2.6.1 Definition of classes. The declarations of local and global objects can include comma-separated lists of initial values enclosed within braces ("{ }"). The following example creates a PathSegment containing an origin of (1,2,3), and a destination of (20,30,0). (The destination z = 0 value is supplied by default.) Within any "{ }" group, an incomplete list of contiguous values can be given. Values cannot be skipped by supplying successive commas.

```
module SampleModule
    {
    class XYZ       // class definition
        {
        double      x, y, z;        // (x,y,z) position
        }

    class PathSegment
        {
        XYZ         origin,         // three objects contained within each PathSegment
                    destination,
                    current_pos;
        double      speed;          // local to PathSegment
        }

    PathSegment MySegment = { { 1.0, 2.0, 3.0 }, { 20.0, 30.0 } };
    }
```

2.4.5 Pointer Operators

Notation	Example	Meaning
pointer -> attribute	p -> speed	The arrow operator is used to access an attribute of an object, given a pointer to the object. The example accesses the speed attribute of the object pointed to by p
object . attribute	MyPath.speed	The "." (period) operator is used to access an attribute of an object, given an instance of the object. The example accesses the speed attribute of object MyPath, where MyPath is a local or global object.

Notation	Example	Meaning
& object	p = &MyPath	The "&" operator yields the address of an object. In this case, pointer p is assigned the address of object MyPath.
** pointer*	*p	The "*" (*dereferencing*) operator yields the object pointed to by a pointer.
(*class* *) *upointer*	(PathSegment*) u -> speed	SLX supports a limited form of what is called *casting* in C and C++. In SLX, casts can be applied to universal pointers (pointer(*)) only. The cast specifies that in the current context, the pointer(*) is to be interpreted as a pointer to a *class* object. Casts are validated at run time. In this example, u must actually point to a PathSegment object.
type(**pointer*)	if (**type**(*p) == **type** widget) Note: if p is a pointer, **type**(p) always = **type pointer**	The **type** operator allows you to determine the type of the object to which a pointer points.

Table 2-17 – Pointer Operators

Note that "*" and "&" are inverses, i.e., "*&x" is the same as "x". (The object pointed to by the address of x is x.)

When an object contains other objects, the arrow and period operators may be used in combination to access attributes of the inner object:

```
PathSegment             MyPath;
pointer(PathSegment)    ps;
double                  CurrentX, CurrentY, CurrentZ;

ps = new PathSegment;
ps -> speed = 10.0;
...
CurrentX = ps -> current_pos.x;     // x-value of current_pos object within PathSegment
```

2.4.6 Use Counts

Every SLX object has a use count. A use count indicates the total number of ways in which an object can be accessed. Consider the following example:

```
pointer(PathSegment)    ps1, ps2;

ps1 = new PathSegment;
ps2 = ps1;
ps1 = NULL;
ps2 = NULL;
```

In the above example, a PathSegment object is created, and a pointer to the object is assigned to ps1. At this point, the use count of the new object is 1, reflecting that there is exactly one way to access attributes of the object, namely, through ps1. When the value of ps1 is copied into ps2, the use count of the object goes up to two, because there are now two ways to access it. When ps1 is assigned a value of **NULL**, the use count goes back to 1, and when ps2 is assigned a value of **NULL**, the use count goes to zero. At this point, the object is automatically destroyed by SLX, because there is no way to access it. This approach solves the "forgotten but not gone" problem, in which unused objects consume memory. The opposite problem, "gone but not forgotten," described below in Section 2.4.7 Destroying Objects, is also solved by means of use counts

Here's what really happens when an assignment is made into a pointer variable:

- The use count of the object to which the pointer currently points is decremented by one.
- The use count of the object to which the pointer points after assignment is incremented by one.

Note that the first point above does *not* say "the object, *if any*, to which the pointer currently points." The implication is that *every* pointer, including **NULL**-valued pointers, points at *something*. ***In SLX,* NULL *pointer values point at a hidden* NULL *object.*** This treatment allows SLX to decrement the use count of an object pointed to by a pointer without having to test whether the pointer is **NULL**. This significantly reduces the cost of use count accounting.

A program can retrieve the use count of an object by using the built-in use_count procedure, which takes a pointer as an argument:

```
pointer(widget)    w;
widget             local_widget;
int                i, j;

w = new widget;
…
i = use_count(w);
j = use_count(&local_widget);
```

2.4.7 Destroying Objects

The following rules govern the destruction of objects:

- When an object is destroyed, the memory that represents the object is made available for reuse.
- An object cannot be destroyed until its use count goes to zero. Destruction of objects with non-zero use counts would be *disastrous*, because a non-zero use count would indicate that there would still be ways to access memory being released. In typical memory management schemes, released memory might remain unused for a while before being allocated for some other purpose. During the interval in which the memory was presumed to be unused, references to a destroyed object would probably go unnoticed. However, if the memory were reallocated, disaster would ensue, because the same memory would be

in use for two different purposes. Destruction of objects with non-zero use counts is called the "*gone but not forgotten*" problem.

- An object is automatically destroyed when its use count goes to zero.
- Global objects are never destroyed. Their inclusion in a module is counted as a use, and since modules persist throughout the execution of a program, the use count of a global object can never be reduced below 1.
- The use count of an object contained inside another object is reduced by 1 when its containing object is destroyed.
- The use count of an object local to a procedure is reduced by 1 when the procedure returns to its caller.
- The **destroy** statement can be used to *explicitly* destroy an object.

The following considerations apply to the **destroy** statement:

- The **destroy** statement takes the following form:

  destroy *pointer_variable*;

- If a **destroy** property has been defined for the object's class, the **destroy** property is invoked. This informs the object that an attempt is being made to destroy it and gives the object the opportunity to remove outstanding uses of the object.
- The **destroy** statement decrements by 1 the use count of the object to which the pointer currently points and assigns a **NULL** value to the pointer.
- If the use count goes to zero, and a **final** property has been defined for the object's class, the **final** property is invoked, notifying the object that its destruction is imminent. In any case, the memory representing the object is released.
- If the use count does not go to zero, and the SLX development environment's "Trap Deferred Object Destruction" option is turned on – it is by default – a warning message will be issued, because the attempt to destroy the object has necessarily resulted in the actual destruction being deferred until the object's use count goes to zero, if ever.

Finally, note that there are some object memory management problems that cannot be solved by use counts. For example, if object A contains a pointer to object B, and object B contains a pointer to Object A, neither object can be destroyed. Such references are said to be *circular*. The destroy property could be used to assign **NULL** values to such pointers in response to attempts to destroy the objects.

2.4.8 Example of the Use of Properties

The example below illustrates the use of SLX properties:

```
module SLXproperties
    {
    pointer(MyObject)    GlobalMptr;    // global pointer

    class MyObject
        {
        static int      TotalObjects;      // static variable shared by all instances of MtObject
        int             ID, j, k;

        initial
            {
            ID = ++TotalObjects;           // count objects, assign ID

            print (ME) "Object _ created\n";
            }

        destroy
            {
            print (ME, use_count(ME))    "_ destroy property invoked, use count = _\n";
            }

        final
            {
            print (ME) "_ actually destroyed\n";
            }

        report
            {
            print (ME, use_count(ME), ID, j)
                "Object _ use count = _, ID = _, j = _\n";
            }

        clear
            {
            j = 0;
            }
        }
    procedure main()
        {
        pointer(MyObject)     LocalMptr;      // Local pointer

        LocalMptr = new MyObject;
        report *LocalMptr;
        LocalMptr -> j = 111;
        GlobalMptr = LocalMptr;               // use count goes to 2
        report *LocalMptr;
        print (LocalMptr) "Destroying _\n";
        destroy LocalMptr;
        print (GlobalMptr) "Removing last use of _\n";
        GlobalMptr = NULL;
        exit(0);
        }
```

The above example produces the output shown below. Note that when SLX prints a pointer, is shows the class name of the object to which the pointer points, followed by the instance number of the object. In this example, there's only one instance of MyObject.

```
Execution begins
Object MyObject 1 created
Object MyObject 1 use count =   1, ID = 1, j = 0
Object MyObject 1 use count =   2, ID = 1, j = 111
Destroying MyObject 1
MyObject 1 destroy property invoked, use count =   1
Removing last use of MyObject 1
MyObject 1 actually destroyed
Execution complete
```

2.4.9 Augmenting the Definition of a Previously Defined Class

All SLX classes, including built-in classes, can be extended by using the **augment** statement. With one exception, the **augment** statement has exactly the same form as a class definition except that the keyword **augment** is substituted for the keyword **class**. The single exception is that an **augment** statement cannot specify arguments to a class. The number and types of class arguments are fixed once a class has been defined.

You can think of the **augment** statement as temporarily placing you back in the context of the original class definition and giving you another chance to specify class attributes and properties. Adding properties is straightforward; all you have to do is make sure the names of the new attributes don't conflict with the names of any existing attributes. On the other hand, you *can* define properties that were previously defined. If you do so, you must be aware of the order in which the original property specifications are merged with the augmented property specifications. Table 2-18 shows the merging rules. Note that in cases where augmented code is placed prior to preexisting code, inclusion of a **return** statement allows you to *completely* replace the preexisting code.

Property	Merge Order	Reason
initial	old, new	Allows you to use initialized variables
final	new, old	Allows you to replace old code.
destroy	old, new	Preexisting cleanup is performed by the time new code is executed
actions	old, new	Your actions must follow preexisting actions
report	new, old	Allows you to replace a predefined report
clear	new, old	Allows you to replace predefined clearing

Table 2-18 – Augment Property Merging

The following example illustrates the use of the **augment** statement:

```
augment MyObject
    {
    double      x;        // additional attribute
    initial
        {
        x = ...;          // additional initialization
        }

    report
        {
        print ...
        print ...
        return;           // skip previously-defined report code
        }
    };
```

2.5 Sets

In many simulation projects, data management issues are often as complex as modeling issues. For example, in large-scale logistics models, keeping track of "where everything is" is a major issue. For such models, SLX sets provide extremely powerful ways to manage collections of objects. In SLX, you can perform the following actions with sets:

- Define *universal* sets that can contain objects of different classes
- Define *homogeneous* sets that contain objects of a single, specified class
- Define ranking criteria for sorted sets
- Place objects into sets
- Remove objects from sets
- Empty sets
- Iterate through the members of a set
- Randomly retrieve objects from sets without costly searches

Sets are frequently used to implement queues. *Ranked* sets can be used to implement complex queuing disciplines.

2.5.1 Defining Sets

Universal sets can contain objects of all classes, and are defined as follows:

```
set(*)          MyUniversalSet;       // a set of objects of any type
```

Homogeneous sets contain objects of a single, specified class, and are defined as follows:

```
set(Widget)     WidgetSet;            // a set of objects of class Widget
```

By default, sets are unranked. Objects can be inserted into an unranked set using **FIFO** and **LIFO** clauses, or **before** and **after** clauses that specify exactly where the object is to be placed, relative to other objects in the set. In the absence of **before** and **after** clauses, objects are inserted into unranked sets in FIFO order.

Both universal and homogeneous sets can be ranked FIFO or LIFO The following are examples of FIFO and LIFO sets:

```
set(Widget)     ranked FIFO  FIFOWidgetSet;       // FIFO set of objects of class Widget
set(*)          ranked LIFO  WidgetSet;           // LIFO universal set
```

before and **after** clauses cannot be used when inserting objects into FIFO or LIFO sets, because their use could violate the declared ordering of the sets.

Homogeneous sets also can be ranked by any combination of attributes of the class of objects that are to be placed in the set. Each selected attribute can be ranked in ascending of descending order. The general form in which attribute-ranked sets are declared is as follows:

set(*class_name*) **ranked**({ { **ascending** | **descending** } *attribute_name* }**,...**)
    *set_name***,... ;**

In the following example, a class named ship is defined with the attributes ship_type and tonnage. The set harbor1 is an unranked set of objects of the class ship. The set harbor2 is a FIFO set of objects of the class ship. The set harbor3 is ranked by ascending ship_type and descending tonnage. When ship objects are inserted into harbor3, their tonnage attributes are examined only in cases in which two ships of the same ship_type are compared.

```
class ship
    {
    enum { freight , passenger , yacht } ship_type;
    double tonnage;
    }

set(ship)                harbor1;           // unranked set
set(ship)  ranked FIFO   harbor2;           // FIFO set

set(ship) ranked(ascending ship_type, descending tonnage)    harbor3;        // ranked set
```

2.5.2 Placing Objects into Sets and Removing Objects from Sets

All set insertion and removal operations are performed by specifying ***pointers*** to the objects to be inserted into or removed from a set. You might find this to be linguistically incorrect. You could argue that language for placing an object into a set or removing an object from a set should specify the object itself, not a pointer to the object. When we designed SLX's set capabilities, we carefully considered the frequency with which various set operations are performed. While local and global objects are often inserted into sets, by far most set insertions and deletions are of dynamically created objects. Dynamically created objects are managed by the use of pointers. If we had chosen to employ syntax in which objects, rather than pointers to objects, were specified in set operations, these operations would have required the use of the dereferencing operator ("*") to specify the object pointed to by a pointer. In other words, typical programs would have had many instances of "perform set operation on **pointer*", and relatively few instances of "perform set operation on *local_or_global_object*". We therefore chose to use "perform set operation on *pointer*" and "perform set operation on &*local_or_global_object*".

Membership in a set provides a way of accessing an object. Therefore, when an object is placed into a set, its use count is incremented by 1, and when an object is removed from a set, its use count is decremented by 1.

The **place into** statement is used to place an object into a set. ***Placing an object into a set of which it is already a member is considered an error.***

```
place  object_ptr  into set_name [ { before | after } reference_object_ptr ] ;
place  object_ptr  into set_name [ FIFO | LIFO ] ;
```

Note that **before/after** clauses and **FIFO/LIFO** can be used only with unranked sets.

The **remove…from** statement is used to remove objects from a set:

```
remove object_ptr from set_name;
```

The following are examples of **place into** and **remove from**:

```
pointer(ship)        Titanic = new ship,
                     QueenMary = new ship;
ship                 LocalShip;

place Titanic into harbor1;
place QueenMary into harbor1 before Titanic;       // explicit, user-specified ranking
place &LocalShip into harbor2;                      // note notation for local objects
remove QueenMary from harbor1;
place QueenMary into harbor1 LIFO;                  // the queen goes to the head of the line
```

2.5.3 Locating Objects in a Set by Position

The first form of the **position** operator returns a pointer to the object at a given position (1…N) in a set. For example,

```
ThirdShip = position(3) in harbor1;
```

assigns to ThirdShip a pointer to the third ship in harbor1. If harbor1 contains fewer members than the specified position, **position** returns a **NULL** value. Note that the **position** operator is ***very slow*** for large sets, since it starts at the beginning or end of a set (whichever is closer to the specified position) and counts objects until it comes to the right one.

The second form of the **position** operator returns the integer position (1…N) of an object in a set, given a pointer to the object. If the object is not in the set, a value of zero is returned.

```
i = position(ThirdShip) in harbor1;
```

Although you could use the above form of the **position** operator to ascertain whether a given object belongs to a set, *don't*! The **position** operator is very slow. See the descriptions of set membership operators in Section 2.5.5 Determining Whether an Object is in a Set.

2.5.4 Locating Objects in a Set Ranked by Attribute Values

SLX includes a **retrieve** operator that retrieves a pointer to an object in an attribute-ranked set, given specified values of the ranking attributes. Consider the following example:

```
module phone_book_demo
    {
    class book_entry
        {
        string(20) last_name;
        string(10) first_name;
        string(20) phone_number;
        };

    set(book_entry) ranked(ascending last_name, ascending first_name)   phone_book;

    procedure main()
        {
        pointer(book_entry)    e;
        …
        fname = "Jim";
        lname = "Henriksen";
        …
        e = retrieve book_entry(first_name=fname, last_name=lname) from phone_book;
        }
    }
```

The above example implements a simplified telephone book as a ranked set. Each object in the telephone book set contains a first name, a last name, and the number corresponding to that name. The set is ranked in ascending order of last name, first name; i.e., people with identical last names are sorted by first name. The **retrieve** operator returns a pointer to a book_entry object in the phone_book set, given a first name and a last name.
The general form of the **retrieve** operator is as follows:

retrieve *class_name* (*attribute_name* = *attribute_value*,**…**) **from** *set_name*

The following rules apply to use of the **retrieve** operator:

- Values must be supplied for all of the set's ranking attributes.
- Values can be *specified* in any order, but they are *processed* in the order in which they were specified in the definition of the set.
- If no object matching the specified attribute values exists in the set, the retrieve operator returns **NULL**.
- Although the above example shows use of the **retrieve** operator in an assignment statement, since it is a full-fledged operator, it can be used in expressions, or in general, anywhere you can use a pointer.

The implementation of the **retrieve** operator is extremely efficient (asymptotically logarithmic complexity). The **retrieve** operator is far more efficient than user-coded searches, so it is the method of choice for finding objects in ranked sets.

Note: SLX includes built-in procedures named rank_root(), rank_lower(), and rank_higher(). These functions are building blocks that underlie the implementation of the **retrieve** operator. Prior to the advent of the **retrieve** operator, these functions were used to implement the equivalent of **retrieve** in user-supplied code. The use of these functions was rather tricky and lead to the implementation of the much easier to use **retrieve** operator. While these functions are still available and supported, you should not use them.

2.5.5 Determining Whether an Object is in a Set

SLX has three Boolean operators for testing set membership:

Syntax	Operation
set **contains** *object_ptr*	**TRUE** if the specified set contains the specified object
object_ptr **is_in** *set*	**TRUE** if the specified object is in the specified set
object_ptr **is_not_in** *set*	**TRUE** if the specified object is ***not*** in the specified set

Table 2-19 –Set Membership Operators

The following are examples of set membership testing operators:

```
if (harbor1 contains titanic)
    ...

if (titanic is_in harbor1)
    ...
```

2.5.6 Iterating Through the Members of a Set

SLX provides two ways of iterating through the members of a set, a collection of low-level operators that allow you to do anything you want, and the **for each** statement, which provides a convenient shorthand that is adequate for many, if not most applications.

2.5.6.1 Set Iteration Operators

SLX provides the following low-level set iteration operators:

Operator Syntax	Meaning
first *class_name* **in** *set*	Returns a pointer to the first object of the specified class in the specified set. If the set is a universal set, objects of a class other than that specified are skipped over.
first object in *set*	Returns a pointer to the first object in the specified set, independent of the object's class. (Used primarily with universal sets)
last *class_name* **in** *set*	Returns a pointer to the last object of the specified class in the specified set. If the set is a universal set, objects of a class other than that specified are skipped over.

Operator Syntax	Meaning
last object in *set*	Returns a pointer to the last object in the specified set, independent of the object's class. (Used primarily with universal sets)
successor(*object_ptr*) **in** *set*	Returns a pointer to the successor of a specified object in a specified set
class_name **successor**(*object_ptr*) **in** *set*	The same as above, except that only objects of the specified class are considered, and objects of other classes are skipped over
predecessor(*object_ptr*) **in** *set*	Returns a pointer to the predecessor of a specified object in a specified set
class_name **predecessor**(*object_ptr*) **in** *set*	The same as above, except that only objects of the specified class are considered, and objects of other classes are skipped over

Table 2-20 – Set Iteration Operators

For each of the operators shown above, if the requested object does not exist, the operator returns a **NULL** value. This is *not* considered an error; however, for the **successor** and **predecessor** operators, if the pointer to the object for which the successor or predecessor is to be determined is **NULL**, an error *will* occur.

The following are examples of low-level set iteration operators:

```
set(ship)          harbor;
set(*)             boats;

pointer(ship)      titanic, queen, ship_ptr;
pointer(*)         boat;

ship_ptr = first ship in harbor;                 // first ship in set
boat  = last object in boats;                    // last object of any kind in set

ship_ptr = successor(titanic) in harbor;         // next ship after titanic
ship_ptr = ship predecessor(queen) in boats;     // skip non-ship objects
```

2.5.6.2 The *for each* Statement

The **for each** statement provides a convenient way to iterate through the members of a set:

```
for (object_ptr  = each { object | class_name } in [ reverse ] set
        [ { before / after | from } pointer_to_an_object_in_the_set ]
        [ with boolean_expression ] )
```

The following are examples of the **for each** statement:

```
for (ship_ptr = each ship in harbor)
for (boat = each object in reverse boats)
for (ship_ptr = each ship in boats from titanic with ship_ptr -> tonnage > 100)
```

The following rules apply to use of the **for each** statement:

- The optional **from** clause specifies the starting point of the iteration.
- The optional **reverse** keyword causes iteration to take place in reverse order, i.e., from the end of the set (or the **from** point) to the beginning.
- The optional **before/after** clauses specify the starting point of the iteration as **before/after** a specified object in the set. Note that the sense of **before/after** is not affected by the **reverse** keyword; i.e., **before/after** clauses are interpreted in the usual sense *prior* to starting a **reverse** iteration.
- If you use **before**, you must also use **reverse**.
- If you use **after**, you cannot use **reverse**.
- When the **for each** statement is used to iterate through the members of a universal set, and a class name is specified, i.e., the **object** keyword is not used, objects of a class other than the specified class are skipped.
- If an entire loop runs to completion, i.e., there is no jump out of the loop, the iteration pointer variable will be **NULL**.
- It is permissible to delete the currently selected object within the scope of the loop. The successor of the currently selected object is determined before the body of the loop is entered, so at the bottom of the loop, the currently selected object is not needed to determine its successor.
- If the predetermined successor to the currently selected object is no longer a member of the set at the bottom of the loop, the entire iteration is restarted from the beginning.

2.5.7 Determining the Size of a Set

The size of a set is stored in a set attribute named size, which is defined as a **control int** variable. One of the most commonplace uses of sets is to implement producer-consumer modeling. In this approach, the producer component of a program creates objects and places them into a set, and the consumer component waits for the size of the set to change and then examines the set, deciding which object, if any, in the set to process (consume). This approach is outlined below:

```
class widget…
pointer(widget)    w;
set(widget)     WidgetQueue;
…
w = new widget;                    // producer actions…
w -> attribute = …;                // set attributes of created widget
…
place w into WidgetQueue;          // place produced widget into a queue
…
wait until (WidgetQueue.size > 0); // consumer waits for something to do
…                                  // consumer actions
```

2.5.8 Emptying a Set

The **empty** statement is used to remove all members from a set:

```
empty [ set ] set_name;
```

The following are examples of the **empty** statement:

```
empty harbor1;
empty set harbor1;          // equivalent to the above
```

Prior to the implementation of the **empty** statement, sets were emptied using the destroy_set() built-in procedure. "destroy_set" is actually a misnomer, because the procedure does not destroy a set; it empties a set. You should not use destroy_set().

2.5.9 Class-Like Properties of Sets

SLX sets straddle the line between a built-in data type and an object class. Treatment of sets as a built-in data type allows SLX to be much more intelligent about set operations than it could be if sets were simply another object class. On the other hand, sets actually are in part implemented as a predefined object class. You can think of sets as a special-purpose class about which SLX has privileged information.

Like any SLX class, sets have properties. The following properties are predefined for sets:

Property	Predefined Behavior
report	A set's **report** property loops through all the objects of the set and executes a **report** statement for each.
clear	A set's **clear** property loops through all the objects of the set and executes a **clear** statement for each.
final	A set's **final** property empties the set.

Table 2-21 – Set Class Properties

As with all SLX properties, you can **augment** any of the above set properties.

2.6 Arrays

SLX includes N-dimensional arrays. There is no bound (other than total memory available) on either the number of dimensions or the extent of each dimension.

2.6.1 Array References

Array references take the following form:

array_name [subscript$_1$] **...** *[subscript$_N$]*

e.g.,

```
MyArray [i+j] [k] [BUSY]
```

2.6.2 Array Declarations

The declaration of an array takes the following form:

> *data_type array_name* [*dimension*$_1$] ... [*dimension*$_N$],... ;

Each dimension may be specified in any of the following formats:

> [*size_expression*]
> [*lower_bound_expression* **...** *upper_bound_expression*]
> [*enumerated_type_name*]
> [*]

In the first form, valid array indices run from 1…size_expression. Note that this convention differs from that of other languages. In particular, it differs from C and C++, where array indices run from 0…size_expression-1.

In the second form, index lower and upper bounds of the dimension are specified explicitly, and the extent of the particular dimension is *upper_bound_expression-lower_bound_expression*+1.

In the third form, the extent of the dimension is determined by the number of enumeration constants defined for the specified enumerated type. Furthermore, any expressions used at run-time as array indices must be variables or constants of the specified enumerated type. The use of integers as indices is prohibited. The enumeration constant **NONE** cannot be used.

In the fourth form, the bounds of the array dimension are inherited from the caller. This form is used only for arguments to SLX procedures or parameterized classes. For array arguments, the number of dimensions must match the number of dimensions of variables passed by the caller, and the bounds of each dimension must be *inherited* ("*").

2.6.2.1 Specifying Initial Values in an Array Declaration

Initial values of array variables are specified as comma-separated lists of values enclosed in braces ("{ }"). For arrays of more than one dimension, each dimension must be enclosed in braces. For each dimension, initial values start with the first array element in the dimension. It is permissible to initialize less than the full extent of any dimension, but individual values cannot be skipped. In other words, successive commas ("value, , value") are not allowed.
Any array elements that are not explicitly initialized are assigned the default initial value for the array's data type, e.g., **NULL** for pointers, zero for integers, etc. **All SLX variables have known initial values.**

The following example illustrates the declaration and initialization of arrays:

```
module ArrayDecls
    {
    int N=4;

    typedef enum { IDLE, BUSY, BROKEN }         MachineState;
    typedef enum { LATHE, DRILLPRESS }          MachineType;

    procedure main
        {
        double x1[10], y1[10][5] ;                  // define two local arrays, no initialization

        int       j1[2][3] = { { 11,12,13 } , {21, 22 } };  // third value of second row missing

        boolean MachineBusy[N] ;                    // extent = a previously defined variable

        int StateCounts[MachineType] [MachineState];        // enumerated type dimensions
        }
    }
```

The following example illustrates the use of arrays as formal parameters of procedures. Procedures are described in Section 2.7 SLX Procedures.

2.6.2.2 Rules for Non-Constant Array Bounds

SLX allows the use of non-constant array bounds, subject to one limitation: any values used as array bounds must have been previously assigned to the variables used in the expressions. ***While this sounds simple, it is not. The major complicating factor is that declarations are processed before executable code.*** Consider the following example:

```
module ArrayDecls
    {
    int N;

    procedure main
        {
        N = 10;
        boolean MachineBusy[N] ;    // ERROR: when this declaration is processed, N = zero
        }
    }
```

In the above example, the setting of N to 10 occurs too late. In this context, N must have a non-zero *initial* value in order for the array declaration to work. One way of getting around this restriction is to use procedures:

```
module ArrayDecls
    {
    int N;

    procedure main
        {
        N = 10;
        MyProcedure(100);
        }

    procedure MyProcedure(int ArrayBound)
        {
        boolean    MachineBusy[N] ;                // OK: global N = 10 (set in main)
        string(20) MyStrings[ArrayBound];          // OK: ArrayBound argument = 100
        }
    }
```

For an example of the use of array procedure arguments, see Section 2.7.4 - Parameter Declarations.

2.6.2.3 Determining the Bounds of an Array at Run-Time

For arrays that have variable bounds, it is sometimes necessary to ascertain the actual bounds during program execution. SLX provides two procedures for doing so:

```
        array_lower_bound(array_name, dimension_number)
        array_upper_bound(array_name, dimension_number)
```

If the specified dimension of the specified array is an enumerated type, the lower bound will always be 1, and the upper bound will be the number of enumeration constants comprising the enumerated type.

Consider the following extension of the example given in Section 2.6.1.3 Rules for Non-Constant Array Bounds:

```
    procedure MyProcedure(int ArrayBound)
        {
        boolean    MachineBusy[N] ;                // OK: global N = 10 (set in main)
        string(20) MyStrings(ArrayBound);          // OK: ArrayBound argument = 100
        int        j, k;
        …
        j = array_upper_bound(MachineBusy, 1);     // upper bound of the first dimension (10)
        k = array_lower_bound(MyStrings, 1);       // lower bound of the first dimension (1)
        }
```

2.7 SLX Procedures

2.7.1 Introduction to Procedures

The syntax and semantics of SLX procedure definitions differ substantially from those of C functions:

- SLX uses wordier syntax for procedure definitions, employing keywords such as **procedure** and **returning** that are not used in C.
- With the exception of DLL procedures (located in externally compiled C or C++ libraries), SLX does not use procedure prototypes. The SLX compiler "sees" all procedure definitions and calls and assures that all calls are compatible with the definitions of the called procedures.
- In C, all arguments are passed by value. This is a one-way form of communication. While C functions are allowed to modify their arguments, changes are not reflected back to the caller of the function.
- SLX procedures allow three forms of parameters: **in** (read-only), **out**, (write-only), and **inout** (read-write). **out** and **inout** parameters are passed by address, so any changes made to procedure arguments are immediately reflected to the caller.

2.7.2 Defining Procedures that do not Return Values

Definitions of procedures that do not return values have the following form:

```
procedure procedure_name( [ parameter_definition,…] )
    {
    declarations…
    executable_statements;
    …
    return;
    }
```

2.7.3 Defining Procedures that Return Values

Definitions of procedures that do not return values have the following form:

```
procedure procedure_name( [ parameter_definition,…] ) returning data_type
    {
    declarations…
    executable_statements;
    …
    return expression;
    }
```

2.7.4 Parameter Declarations

Declarations of procedure parameters take the same form as "ordinary" declarations, with the following exceptions:

- The prefixes **in**, **out**, and **inout**, defined in Section 2.7.1 Introduction to Procedures, can be used. If none of these prefixes are used, a parameter is assumed to be **in** (read-only).
- Array arguments must specify dimensions as "*" (inherited from the procedure's caller).
- String arguments must specify maximum lengths of "*" (inherited from the caller).

The following example illustrates the use of procedures:

```
module fibo
    {
    procedure fibonacci(in int argument) returning int
        {
        if (argument <= 2 )
            return 1;
        else
            return fibonacci(argument – 1) + fibonacci(argument - 2) ;
         }

    procedure main
        {
        int          parameter, result ;

        parameter = 30;
        result = fibonacci (parameter);
        print (parameter, result)  "Parameter = _  Result = _\n";
        exit(0);
        }
    }
```

The following example illustrates the use of array arguments:

```
procedure InitializeArray(inout int z[*] , int N)
    {
    int i;

    for (i=1;i<=N;i++)
        z[i] = i;                        // assign values to the first N array elements
    }

procedure main
    {
    int i , x1[10];

    InitializeArray(x1 ,8);              // set the first 8 values of x1

    for (i=1;i<=10;i++)
        print( i , x1[i] ) "_ : _     ";   // print all 10 values of x1
    }
```

The following example illustrates the use of string arguments:

```
procedure MakePlural(inout string(*) z)         // z's length is inherited
    {
    z cat= "s";
    }

procedure main
    {
    string(20)          x = "widget";

    MakePlural(x);      // Result: x = "widgets"
    }
```

If you invoke SLX from a command line or from another program, arguments can be passed to an SLX main procedure. The main procedure must take the following form:

```
procedure main( int argc, string(*) argv[*])
```

SLX breaks the invocation command line into blank-delimited tokens. If you need to include blanks in an input token, you must enclose it in quotes. The first argument, *argc*, is the number of tokens passed. It can be zero. The second argument, *argv*, is an array of string variables, one for each token, starting with *argv*[1]. (*argv*[0] is set to the name of the root file of the program being run.) SLX determines the size of the array (the number of tokens) and the length of each.

2.7.5 SLX Methods

2.7.5.1 What's an SLX Method?

An SLX method is an SLX procedure conceptually (if not physically) contained within an SLX class. In the following example, "inci" is a method of the class "widget":

```
class widget
    {
    int i;

    procedure inci(int increment)
        {
        i += increment;
        return;
        }
    };
```

2.7.5.2 Advantage #1 of Using Methods

Within a method, elements of the method's containing class can be used without a qualifying pointer. This is possible for two reasons. First, SLX always tries to find the declaration for a variable by first searching the current scope and any outer, containing scopes. When the compiler sees "i" within "inci", it first looks for a declaration of "i" within "inci". Finding none, it searches the outer scope of "inci", which is the scope of "widget". Here, it finds a declaration for "i". Thus, SLX knows that "i" within "inci" is an element of "widget". Second, when a method is called, SLX passes a hidden zeroeth argument that is a pointer to the instance of the class that contains the method. This hidden pointer is used to access elements of the class.

The alternative would be to pass an explicit pointer argument identifying the object to be operated upon:

```
procedure inci(pointer(widget) w, int increment)
    {
    w -> i += increment;
    return;
    }
```

This procedure could be placed anywhere. It needn't be placed within class widget, since the only reference it makes to an element of class widget is explicitly qualified

2.7.5.3 Calling Methods

2.7.5.3.1 Calling a Method from Outside its Class

To call a method from outside its class, one must prefix the method name with a pointer or "dot" prefix that identifies the particular object on which the method is to operate:

```
procedure main()
        {
        pointer(widget)    w;
        widget             localw;

        localw.inci(2);        // local object; "dot" qualification

        w = new widget;        // dynamically created object; pointer qualification
        w -> inci(3);
        }
```

In the above example, we see inci applied to a local instance of class widget, named "localw" and to a dynamically created instance pointed to by "w".

2.7.5.3.2 Calling a Method from Within its Class

When a method is called from within its class, you needn't qualify the method name with a prefix, since SLX assumes the method is to be applied to the current instance of the class. Alternatively, you can qualify such a method call by using **ME**. Assume that class widget included the following actions property:

```
actions
    {
    inci(5);
    ME -> inci(5);        // unnecessary qualification (equivalent to the above notation)
    }
```

The two approaches shown above are functionally equivalent. In the unqualified approach, the passing of **ME** is implicit. In both cases, **ME** is passed as a hidden, zeroeth argument that does not appear in inci's definition.

2.7.5.4 Using the Same Method with More than One Class

Suppose you're building a model of a traffic grid, and your model contains classes for cars, trucks, and buses. Further suppose that you wish to construct an "EnterGrid" method for each of these classes. EnterGrid performs an identical function for each of the three classes, but because the objects entering the grid differ from one another, the implementations of each of the three versions of EnterGrid may differ considerably.

You can call such methods using either class-specific pointers or universal ("pointer(*)") pointers:

```
pointer(Car)        c;
pointer(Bus)        b;
pointer(Truck)      t;
pointer(*)  u;

c = new Car;
b = new bus;

c -> EnterGrid(…);      // method known to be a "car" method
u = b;
u -> EnterGrid(…);      // pointer(*) => run-time method selection
```

The first call of EnterGrid shown above is easy for SLX to interpret, since it's obvious that the version of EnterGrid to be called is the version defined for class Car. The second call is more difficult. SLX must examine the object to which "u" points, and select the version of EnterGrid for the object's class. SLX is able to do so in a fixed sequence of instructions. (Method selection is accomplished by a direct lookup, not a search.) Thus, the overhead of method selection is extremely low.

To use methods specified by universal pointers, the following rules apply:

- The methods must be identically named.
- The methods must have identical argument lists.
- The methods must all return identical types (or all have *no* return values).

The following errors can occur when calling a method specified by a universal pointer:

- The pointer may point to a class that doesn't include the method.
- The pointer may be **NULL**.

SLX traps both of these errors.

2.7.5.5 Advantage #2 of Using Methods

Specifying methods by means of universal pointers eliminates the need for manual method selection; i.e., you do *not* have to do the following:

```
if (type(*ptr) == type Car)
        (Call the Car method)

else if type(*ptr) == type Truck)
        (Call the Truck method)
```

When you specify a method by using a universal pointer, you achieve an ideal notation: your intent is clear, while implementation details are hidden. When you look at

```
    u -> EnterGrid(…)
```

You can see the intent without having to worry about details.

2.7.5.6 Defining Methods Outside Their Classes

SLX provides two ways of defining methods outside their classes. However, you should think of both approaches as temporarily placing the method being defined back into the context of the class in which it is conceptually, but not physically contained. In other words, even when a method is defined outside its class, it acts as if it were physically contained within its class.

The first approach is to use SLX's rarely used **augment**, which allows you to add things to a class after the class has been defined. For example, if we had not included method inci in class widget, we could do so as follows:

```
augment widget
        {
        procedure inci(int increment)
            {
            …
            }
        }
```

The **augment** keyword temporarily places your program back in the widget class. Note that augment allows you to do considerably more than add methods. For example, you can add elements to a class; you can add a property, e.g., **initial**; or you can extend the code of existing properties.

The second approach is to use OOP-inspired notation:

```
procedure widget::inci(int increment)
    {
    …
    }
```

In this notation, "widget**::**" accomplishes a sufficient subset of "augment widget", using a notation known outside the wild, wonderful world of SLX.

Methods for classes defined within precursor modules ***must*** be included within the precursor module. In other words, once a precursor module has been compiled, augmenting its classes by adding methods outside the precursor module is not allowed.

2.7.5.7 Pseudo-Methods

To qualify as a true method, a method must include unqualified references to elements of its containing class. However, it's possible to code methods that include no such references. Such methods are pseudo-methods, since they look like ordinary methods and are called like ordinary methods, but have no "real" requirement for specifying which object of a given class they are to operate upon, because they contain no implicit references to their classes.

For historical reasons, SLX is lenient in its handling of pseudo-methods. It allows you to call them without pointer or "dot" qualifications; i.e., you can call them as ordinary SLX procedures.

2.7.5.8 Debugger Support for Methods

Within a method, the SLX debugger's Local Data (Dynamic Selection) window contains a pseudo-variable designated as "(*class name*)". For example, if you step into the inci method, the Local Data window will contain a pseudo-variable named "(widget)". By right clicking on "(widget)", you can explore the particular widget currently operated upon by inci.

2.7.5.9 A Comprehensive Example of Methods

```
public module Methods1
    {
    class widget
        {
        int     i;
        double    x;

//      procedure inci(int increment, int incompatibility)      // incompatible with widget2's inci
//      procedure inci(int increment[*])                        // incompatible with widget2's inci
//      procedure inci(int increment, pointer(widget) w)        // incompatible with widget2's inci

        procedure inci(int increment)
            {
            i += increment;
            return;
            }

        procedure fetch_i returning int
            {
            return i;
            }
        };
    }                   // end of module Methods1

module Methods2
    {
    class widget3
        {
        int     i, inci;
        };

    class widget2
        {
        int     i;
        double    x;

//  procedure inci(int increment, pointer(widget2) w2)        // incompatible with widget's inci

    procedure inci(int increment)
            {
            i -= increment;
            return;
            }
        };
    procedure widget2::fetch_i returning int                // OOP-style
        {
        return i;
        }

    augment widget2
        {
        procedure store_i(int j)
            {
            i = j;
            }
        }

    procedure main()
        {
        pointer(widget)    w;
        pointer(widget2)   w2;
        pointer(*)  u;
        int  i;

        widget      localw;
        widget2     localw2;

        localw.inci(2);      // dot-style qualification; method known from object type
        localw2.inci(2);

        w = new widget;
        w2 = new widget2;

        w -> inci(3);        // arrow-style qualification, class known from pointer type
        w2 -> inci(3);

        u = w;
        u -> inci(3);        // universal pointer used; method must be determined
        i = u -> fetch_i();

//      u -> store_i(27);    // run-time error: class widget has no store_j method

        u = w2;
        u -> inci(3);
        i = u -> fetch_i();

        u = NULL;
//      u -> inci(3);        // run-time error: NULL pointer

        u = new widget3;
//      u -> inci(3);        // run-time error: widget3 class has no inci method
        }
    }
```

2.8 Control Flow Statements

A language's control flow statements specify the order in which computations are performed. SLX's control flow statements are closely modeled on those of C. In some cases, SLX statements are restricted versions of their C counterparts, and in other cases, SLX goes beyond the capabilities of C.

2.8.1 Statements and Compound Statements

The C language, on which much of SLX's syntax is modeled, is a so-called expression-based language; i.e., expressions terminated with a semicolon can be used as statements. For example,

```
i + j;
```

is a valid statement; however, since the result of the addition is not stored, it accomplishes nothing. SLX is for the most part a statement-based language. In SLX, the C statement shown above is illegal. The only cases in which expressions can be used as statements are shown below:

```
++j;                    // preincrementing
--j;                    // predecrementing
j++;                    // postincrementing
j--;                    // postdecrementing
MyProcedure(x, y, z);   // free-standing procedure call
new widget;             // free-standing object creation (See note below)
```

Note that free-standing object creation must somehow create a reference to the created object; otherwise, the object will be created and immediately destroyed, and this is considered an execution error. The only way to accomplish this is via the **initial** property of the object's class. For example, the **initial** property might place all created objects into a set.

In SLX, all forms of assignment operators are considered to be statement-level operators, rather than expression-level operators. For example,

```
j += 10;
```

is an assignment statement, not a free-standing expression. Because C considers assignment operators to be expression-level operators, it's possible to do the following in C:

```
j = k += 10;
j = k = 10;
```

In SLX, these constructs are illegal, largely because the use of embedded assignments can present pitfalls to beginning users.

A *compound statement* is a sequence of zero or more statements enclosed in braces ("{ }"), e.g.,

```
{
j = k;
k += 100;
}
```

A compound statement can be used anywhere a statement is required. A few SLX constructs require the use of compound statements.

2.8.2 if…else

The general form of **if…else** is as follows:

```
if (Boolean_expression)
        statement1;
[ else
        statement2; ]
```

statement1 is executed only if *Boolean_expression* evaluates to **TRUE**. If *Boolean_expression* is **FALSE**, and the optional **else** clause is specified, *statement2* is executed. The following is an example of **if…else**:

```
if ( a>b)
        a=+10 ;
else
    {
    a++;
    b *= a;
    }
```

2.8.3 The switch Statement

The **switch** statement is used to select statements to be executed based on the value of a *case selector* expression. The general form of the **switch** statement is as follows:

```
                switch ( case_selector_expression )
                        {
case expression1:       statements
...
case expressionN:       statements

[ default:              statements ]
                        }
```

The **switch** statement operates as follows:

- *case_selector_expression* is evaluated.

- For data types other than enumerated types, the value of the case selector expression is compared to the **case** label expressions, one-by-one, until a match is found. If a match is found, program execution jumps to the **case**-labeled statement.
- For enumerated type expressions, SLX constructs a jump table, since the range of expression values is known in advance. Case selection for enumerated types is very fast.
- In the absence of **break** statements (See Section 2.8.6 The continue and break Statements) execution flows from one **case** into the next, if any. In other words, reaching the end of a **case** does not imply a jump to the end of the **switch** statement.
- If no matching **case** if found, and an optional **default** label is specified, execution jumps to the statement with the **default** label.
- If no match is found, and no **default** label is specified, no cases are executed.
- Duplicate constant case labels are not allowed.

Unlike C, case selector expressions can be of any data type, and case label expressions can be full-blown expressions. They need not be constants. The following example shows two uses of the **switch** statement that go beyond the capabilities of C:

```
    string(10)      keyword;
    double          x, y;
    ...
    keyword = "large";
    switch (keyword)        // character string case selector
                    {
case "small":       ...
                    break;
case "medium":      ...
                    break;
case "large":       ...
                    break;
                    }

        switch (TRUE)       // boolean case selector (very unusual)
                    {
case x < y:         ...
case x > y:         ...
default:            ...
                    }
```

2.8.4 The goto Statement

The **goto** statement is used to jump to an arbitrary label. A label can be placed on any executable statement. The following example illustrates the use of the **goto** statement:

```
          if (x < 0.0)
               goto badx;
          ...
badx:   print (x)     "x = _.___ (less than zero)\n";
```

2.8.5 Loops

2.8.5.1 The General-Purpose for Loop

The general form of the general-purpose for loop is as follows:

for (*statement1*; *boolean_expression*; *statement2*)
  body_statement;

The general-purpose **for** loop operates as follows:

- *statement1* is executed.
- *boolean_expression* is evaluated. If it is **TRUE**, *body_statement* is executed; otherwise, execution jumps to the first statement following *body_statement.*
- At the "bottom" of the loop, *statement_2* is executed, and execution jumps back to evaluation of *boolean_expression*.

C users should note that SLX's version of the **for** statement is considerably restricted in comparison to C's.

The following is an example of the **for** statement:

```
int i, j;

for (i=1; i<=10; i++)              // i = 1, 2, …, 10  (On exit from the loop, i = 11)
     j = i * i ;
```

2.8.5.2 The while Loop

The **while** loop allows repeated execution of a statement for as long as a specified condition is **TRUE**. The general form of the **while** statement is as follows:

while (*boolean_expression*)
  statement;

If the specified condition is initially **FALSE**, the body of the loop is never executed. The following is an example of the **while** loop:

```
int i, j;               // i is initialized to zero

while (i < 10)         // i is incremented each time through the loop (final value == 11)
     {
     i++;
     j = i*j;
     }
```

2.8.5.3 The do…while Loop

The **do…while** loop allows repeated execution of a statement for as long as a specified condition is **TRUE**. The **do…while** loop is similar to the **while** loop, except that the specified condition is tested at the bottom of the loop, so ***a* do…while *loop is always executed at least once***. The general form of the **do…while** statement is as follows:

```
     do
          statement;
     while  (boolean_expression);
```

Omission of the semicolon at the end of the **while** clause is a commonplace error, even among experienced programmers. Sooner or later, you too will do this. The following is an example of the **do…while** loop:

```
int i, j;               // i is initialized to zero

do
     {
     i++;
     j = i*j;
     }
while (i <= 10);
```

2.8.5.4 The forever Loop

The **forever** loop is a loop that executes until explicitly exited by a **goto** or **break** statement. The general form of the **forever** loop is as follows:

```
     forever
          statement;
```

The following is an example of the **forever** loop:

```
int i, j;                    // i is initialized to zero

forever
    {
    i++;
    j = i*j;

    if (i > 10)
        break;               // exit the forever loop
    }
```

2.8.5.5 The for each Loop

There are two forms of the **for each** loop, one for iterating through the members of a set and one for iterating through the types of an enumerated type. The former is discussed in Section 2.5.6 Iterating Through the Members of a Set and the latter in Section 2.3.4.3 Enumerated Type Statements.

2.8.6 The continue and break Statements

The **continue** statement can only be used within a loop. It causes execution to jump to the next iteration of the loop.

The **break** statement can be used both within loops or within **switch** statements. The **break** statement applies to the innermost loop or **switch** statement enclosing it. When used in the body of a loop, the **break** statement causes execution to jump out of the loop. When used in a **switch** statement, the **break** statement causes execution to jump just past the end of the body of the **switch** statement.

The following is an example of the **continue** and **break** statements::

```
int i, j ;

forever
    {
    i++;
    if (i % 2 == 0)          // skip even values of i
        continue;
    else
        j = i;

    print ( j) "_\n";
    if (i > 10)
        break;               // exit the loop
    }
```

2.8.7 Transferring Control to the SLX Debugger – SLXDebugger()

The SLXDebugger() procedure allows a program to yield control to the SLX Debugger, passing a message to be displayed. The debugger is entered, and the next statement to be executed is highlighted. The following example illustrates use of the SLXDebugger() procedure:

```
    SLXDebugger("This program is in trouble.");
```

2.8.8 The exit() Procedure

The SLX exit() procedure ends the execution of a program. When SLX is invoked from a command line or from another program, the argument to the exit() procedure is returned to the operating system or the caller, respectively. The following is an example of the use of the exit() procedure:

```
    exit(-99);   // error return
```

Note that if execution flows off the end of an SLX main program, an implied "exit(0);" is issued.

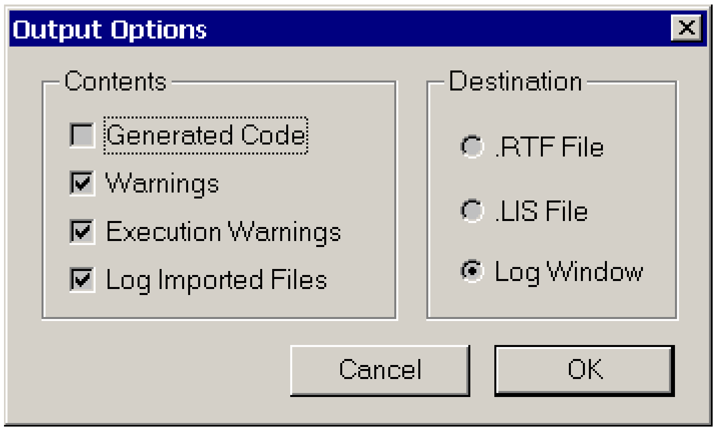
2.9 Input / Output

2.9.1 SLX Files

Information about SLX files is stored in slx_file objects. While it is in theory possible to manipulate slx_file objects directly, most users should use macros and built-in procedures SLX provides for this purpose. Lower-level procedures for input/output are intentionally undocumented. While you can readily ascertain the names of such procedures and see how SLX uses them by looking at expansions of SLX-provided I/O macros, you do so at your own risk.

2.9.1.1 Predefined SLX Files

SLX provides three standard files, stdout, stderr, and stdin. By default, output to stdout is written to the log window of SLX's development environment. Clicking on Options, Output will cause a dialog box of the following form to pop up:


Output Options

Contents
- ☐ Generated Code
- ☑ Warnings
- ☑ Execution Warnings
- ☑ Log Imported Files

Destination
- ○ .RTF File
- ○ .LIS File
- ● Log Window

You can change the destination of stdout output by selecting the .LIS File or .RTF Fle options. If you select either of these options, the file name to which output is written is the same as the name of the root file of the SLX program being executed. If you are invoking SLX from a command line (See Section ?.?), you can use the /output command line option to specify the destination of stdout output.

The use of RTF files is described in Section 2.9.2.4 RTF output.

Output to stderr is written to SLX's log window.

The predefined file stdin is used to read input from the keyboard.

2.9.1.2 Defining Your Own SLX Files

SLX files are defined by using the **filedef** statement. The general form of the **filedef** statement is as follows:

```
filedef  internal_name
         { [ name = external_name ]
           [ options = option,... ]
           [ buffer_size = size ]
           [ window = left, right, top, bottom ]
           [ input / output ]
         } * ;
```

internal_name is used as the name of an slx_file object. *external_name*, frequently specified as a character string constant, is the name of a disk file. If you use a character string constant to specify a file name, and the file name contains backslash ("\") characters, remember that you must double any backslash characters, e.g., to specify a file named C:\MyRoot\Project1\main.slx, you must specify "C:\\MyRoot\\Project1\\main.slx".

The following options can be specified for SLX files:

Option	Meaning
append	If an output file already exists, the file's output pointer is positioned to the end of the file, so any additional output is appended to the file. If an output file already exists, and the **append** option is *not* used, the file will be emptied and rewritten from the beginning.
unbuffered	An output file is written fragment-by-fragment. Multiple writes can occur within an output line.
line_buffered	Lines of output are written only when a complete line image has been constructed in memory.
rtf	File is output in RTF (Rich Text) format. (See Section 2.9.2.4 RTF output.)
newline	If used with an input file, each **read** statement that references the file will start by reading a new record. If the **newline** option is not specified for an input file, each **read** statement continues by reading the next input item, if any, after the previously read input item. This is called *stream* input.

Table 2-22 – Filedef Options

The following are examples of the filedef statement:

```
filedef MyInput    name = "c:\\MyProject\\InputData1.dat" input options=newline;
filedef MyOutput  name = "c:\\MyProject\\Output.lis" output options=append;
```

If you are reading or writing files whose record lengths exceed 4096 bytes, you must specify an appropriate **buffer_size** value. If you fail to do so, input records will be truncated at 4096 bytes, and output records will be wrapped at 4096 bytes; i.e., newlines will be inserted.

If you wish to write output to a window, rather than a file, you can specify **window** dimensions. Each of the four values is interpreted as a fraction of screen width or height, e.g.,

```
    window = 0.5, 1.0, 0.5, 1.0
```

creates a window in the lower right quadrant of the screen.

The optional **input** and **output** keywords can be used to specify a file's purpose. If neither option is specified, a file's mode of operation is determined by the first operation (**read**, **write**, or **open**) performed. Note that files can be written, closed, and read as many times as necessary; however, a file designated as an **input** file cannot *ever* be written, nor can an **output** file be read.

2.9.2 SLX Output

SLX programs can write output to four classes of destinations:

- Output can be written to the predefined files stdout and stderr.
- Output can be written to files defined by using the **filedef** statement.
- Output can be written to a destination string (in-memory output).
- Output can be written to the compiler's input stream. Used for macro expansion, this special form of output is discussed in Section 7.?.

The following data types can be written:

- int
- double
- string
- boolean
- enumerated type
- pointer
- type(expression) (typically type(**pointer_variable*))

SLX writes pointer values as the symbolic name of the object to which a pointer points. If the object is a dynamically created object, SLX writes the name of the object's class, followed by an integer instance ID; e.g., the 27th widget object would be written as "widget 27". If a pointer is **NULL**, SLX writes "NULL".

2.9.2.1 Picture Format Processing

All SLX output is written using ASCII "picture" formats. SLX does not provide for binary output. The following rules apply to pictures:

- A picture is comprised of three types of text:
  - Literal text is copied directly to the output file.
  - *Substitution fields* specify formats into which program data is edited.
  - *Escape sequences*, e.g., "\n", specify line spacing and special effects.
- If edited data consumes more characters than have been specified in a picture, the remainder of the picture is shifted to the right; i.e., output is squeezed into the picture.
- Integer and floating point data are by default written right-justified. All other data types are by default written left justified.
- Justification of a substitution field can be specified explicitly by using "|" characters. (Examples are given in Table 2-23, below.)
- If a decimal point is specified in a substitution field, and the data edited into the substitution field is not floating point data, the decimal point is ignored.
- If no decimal point is specified in a substitution field, and floating point data is edited into the substitution field, the data is rounded to the nearest integer.

The preferred character for defining a substitution field is the underscore ("_"); however, SLX also accepts the asterisk ("*") character for this purpose. For SLX macro and defined statement picture output (See Section 7.?), the pound sign ("#") is used. In most typefaces, the underscore character is as wide as the widest characters, and the asterisk character is not. Since the purpose of an SLX picture is to represent the appearance of program output, underscore characters are preferable to asterisks, because they reduce the probability of having to squeeze edited text into a picture, shifting remaining text to the right.

Table 2-23 shows escape sequences and representative substitution fields used in SLX pictures:

Characters	Meaning
_____	Substitution field using default justification for the data type edited into the picture
|_____	Left-justified substitution field
_____|	Right-justified substitution field
___|___	Centered substitution field
_____.___	Floating point substitution field (decimal point determines alignment)
\"	Allows inclusion of a quotation mark in a picture. (Without the "\", a string constant would end.)
\\	Allows inclusion of a backslash character in a picture.
\f	An ASCII formfeed character
\n	A newline character (used to terminate a line of output)
\xhh	The hexadecimal value "hh" is inserted.

Characters	Meaning
\B	Starts/ends text to be written in boldface
\I	Starts/ends text to be written in italics
\R	Starts/ends text to be written in a red color
\U	Starts/ends text to be written with underscores
_	Allows inclusion of an underscore ("_") in a picture
*	Allows inclusion of an asterisk ("*") in a picture
\#	Allows inclusion of a pound sign ("#") in macro-generated text

Table 2-23 – SLX Picture Components

The following example illustrates the use of substitution fields:

```
double          x ;

for (x = 1.0; x <= 10000.0; x *= 10.0)
        print (x, x, x, x) "|_______ _______| ___|___ ______._\n";
```

The code shown above produces the following output:

```
1            1           1            1.0
10          10          10           10.0
100        100         100          100.0
1000      1000        1000         1000.0
10000    10000       10000        10000.0
```

2.9.2.2 Multi-Line Pictures

Many pictures specify more than one line of output. You can specify multi-line output in two ways. First, you can construct one long picture that contains newline ("\n") escape sequences where you need line breaks, e.g.

```
    "Line1\nLine 2\nLine 3\n"
```

If the individual lines are relatively short, this compact notation may be convenient. To specify multi-line pictures for relatively long lines, it's easier to use the following approach:

```
    "This is long output line #1"
    "This is long output line #2"
    "This is long output line #3\n"
```

Note that only the last line includes a newline escape sequence. The reason for this is that the picture comprises three string constants, and when SLX processes consecutive string constants, it combines them into a single string constant, ***automatically inserting newlines*** at the points at which string constants were glued" together. ***Note that this behavior is incompatible with C, which does* not *insert newlines.***

2.9.2.3 Special Effects in Pictures

Table 2-23 (above) includes escape sequences for inserting special effects into slx pictures. The following example illustrates the use of some of these escape sequences:

```
"Line 1 includes \B\Rboldface red\R\B text"
"\_Line 2 is underlined in its entirety\_\n"
```

Special effects of the type shown above are ignored when output is written to plaintext files.

2.9.2.4 RTF output

Output that is written to the SLX development environment log window and to files with an RTF extension is written in RTF (Rich Text Format). RTF output can include boldface, italic, and underscored text, and it can contain red-colored text. For example, if you wish to write an error message, you might choose to write it in boldface red. RTF output can be readily cut and pasted using Microsoft Word-compatible tools.

2.9.2.5 Output to stdout

The predefined file stdout is by default routed to the log window of SLX's development environment. By clicking on Options, Output, you can change the destination of stdout to be *filename*.lis or *filename*.rtf, where *filename* is the root file of the SLX program being executed. For command-line invocations of SLX, the /output option can be used to specify a file to which stdout output is written.

SLX provides the **print** statement for writing to stdout. (**print** is equivalent to **write file**=stdout.) The general form of the **print** statement is as follows:

print [**options** = *style*,...] [(*expression* ,...)] *picture* ;

The following optional styles can be specified:

Style	Meaning
bold	Text is written in boldface.
italic	Text is written in italics.
red	Text is written in a red color.
underline	Text is underlined.

Table 2-24 – Picture Styles

The following is an example of the **print** statement:

```
print (x, sin(x))     "x = _.____  sin(x) = _.____\n";
```

Note that within a picture, the styles specified in a **print** statement can be overridden, e.g.,

```
print options=bold "Mostly boldface and \Bsome non-boldface\B text\n"
```

The output of the above statement is

Mostly boldface and some non-boldface **text**

One or more comma-separated expressions can be provided in an optional output expression list. If any expressions are provided, they are edited one-by-one into successive substitution fields of the mandatory *picture*. If there are more output expressions than substitution fields, excess expressions are edited using default formats and appended to the picture-specified output. If fewer expressions than substitution fields are provided, the excess substitution fields are written literally (without substitution).

2.9.2.6 Output to Files

The **write file**= statement is used for writing output to files. The general form of the **write file**= statement is as follows:

write file = *internal_filename* [**options** = *option,…*] [(*expression ,...*)] *picture* ;

internal_filename is the name of an slx_file object within the SLX program.

The *options* of the **write file**=statement are identical to those of the **print** statement, discussed in Section 2.9.2.5 Output to stdout.

The following is an example of the **write file**= statement:

```
filedef OutFile name="Output.lis";

write file=OutFile (x, sin(x))   "x = _.____  sin(x) = _.____\n";
```

Note that **write file=stdout** is equivalent to **print**.

2.9.2.7 String Output

Most SLX output is written either to the screen or to files; however, in some situations, it's useful to write output to an SLX string. For example, it might be easier to construct a complicated string using output operations than it would be to construct individual pieces of the string and combine the pieces using concatenation operations. Another application of string output is to immediately follow a write to a string with a read from the same string. This technique can be used to accomplish data conversions that are not built into SLX.

Output is written to strings using the **write string**= statement, which has the following general form:

write string = *string_variable* (*expression,...*) *picture*

Output is written to the specified *string_variable*. Optional expressions can be edited into the string output in accordance with the mandatory *picture*. If the length of the edited output exceeds the length of the target string, the edited output is truncated. This is *not* treated as an error.

The following is an example of **write string**=:

```
string(20) city = "Alexandria",
           state = "VA",
           zip = "22305";

string(50) address;

write string=address (city, state, zip)    "_, _ _";    // address = "Alexandria, VA 22305"
```

2.9.2.8 Diagnostic Output

SLX generates many of its error and warning messages using the **diagnose** statement. You can use the **diagnose** statement for the same purpose. For example, suppose that you're responsible for writing an SLX-based software package for use by others. If the package is going to be heavily used, you might find it worthwhile to detect some error conditions and generate package-specific diagnostic messages for them, rather than allowing SLX to trap the errors and generate generic messages. Preventing division by zero is a good example. The error will be trapped no matter what you do, but trapping it yourself is more graceful than allowing SLX to do so.

The **diagnose** statement can be used to issue warning messages or error messages at compile-time or run-time. Run-time errors terminate program execution. Compile-time messages can be issued only during the processing of macros and statement definitions (See Section 7.?).

Execution of a **diagnose** statement always causes a message to appear in a source code window at the *point of occurrence* of the error, *not* the point at which the error was *discovered*.

Every **diagnose** statement must identify at least one source code item to be highlighted in red. A **diagnose** statement that is issued in a macro/statement definition or a procedure can specify that its **caller** is faulty, and the entire invocation of the macro/statement or procedure will be highlighted in red. In contexts where more specific information is available, the **diagnose** statement can specify one or two (but at least one) specific source code items (typically expressions used as procedure or macro/statement arguments).

The format of the message issued by a **diagnose** statement is determined by a picture. Processing of the picture is identical to that of ordinary SLX output pictures, with one exception. Within **diagnose** picture, the caret character ("^") is used to define substitution fields into which faulty source code items are edited. A picture can have at most two such substitution fields. Each must correspond to a faulty item specified in the **diagnose** statement.

The general form of the diagnose statement is as follows:

```
diagnose

    [ { what1 [, what2 ] } | caller ]
    [ compile_time / run_time ]
    [ error / warning ]
    [ ( picture_arguments ) ]        picture ;
```

If a **diagnose** statement is executed within a procedure or class, *what1* and *what2*, if used, must specify arguments of the procedure / class.

If a **diagnose** statement is executed within a macro or statement definition, *what1* and *what2*, if used, must specify arguments of the macro / statement.

If *what1* and *what2* are supplied, source code corresponding to them is highlighted at the point of macro/statement/procedure invocation. If *picture* contains caret characters, *what1* is substituted for the first, and *what2* for the second. If not, *what1* and *what2* are only highlighted in source code and not edited into the message generated by the **diagnose** statement.

There are no restrictions on the number of optional *picture_arguments* that can be specified. Substitution fields in the picture for these arguments are defined as normal substitution fields; i.e., using underscore characters.

The following examples illustrate the use of a diagnose statement in a procedure:

```
procedure MyProc(double a, double b)
    {
    if (a < b)
        diagnose caller run_time warning
            "The first argument of MyProc is less than the second";
    }
```

```
procedure MyProc(double a, double b)
    {
    if (a < b)
        diagnose caller run_time warning (a, b)
            "The first argument of MyProc (_.__) is less than the second (_.__)";
    }
```

```
procedure MyProc(double a, double b)
    {
    if (a < b)
        diagnose a, b run_time warning (a, b)
            "^ (_.__) is less than ^ (_.__)";
    }
```

Assume that MyProc is called with a first argument less than the second argument. In the first example, the entire offending call of MyProc will be highlighted in red in its source window, and an informative, but very generic message will appear beneath the source code.

In the second example, the values of the arguments are edited into the message. This is a considerable improvement.

In the third case, only the arguments to MyProc are highlighted in the offending call. Furthermore, the actual expressions supplied as arguments are edited into the message. A sample offending invocation and the resulting message are shown below.

```
MyProc(1.0 + 2.0, 3.0 + 4.0);
•• Execution warning at time 0: "1.0 + 2.0" (3.00) is less than "3.0 + 4.0" (7.00)
```

2.9.3 SLX Input

The following data types can be read from the keyboard, from files, and from string variables:

- int
- double
- string
- boolean
- enumerated type

Input data items must be separated in any of the following ways:

- one or more blanks
- a TAB character and zero or more blanks
- a comma and zero or more blanks

If input to a string variable contains blanks, the input data item must be enclosed in quotes. If input to be read into a string variable is longer than the maximum length of the string variable, it is truncated. This is *not* treated as an error.

By default, SLX uses stream input. When stream input is used, any given input request from a given file will start by reading any data remaining from the previous input request from the same file. Options described below allow overriding this default behavior. If input is read into a list of variables, as many records are read as necessary to read data into all variables in the list.

2.9.3.1 Reading from Files and the Keyboard

Input can be read from a file or from the keyboard using a **read file**= statement, which has the following form:

```
read    [ newline | record ] file = internal_filename
        [ prompt = prompt_text ]
        [ end = end_label ]
        [ err = error_label ]
        (variable,... ) ;
```

The following options are available:

Option	Explanation
newline	The **newline** option causes a **read** statement to start by reading a new record. Unused data in a previously read record, if any, is discarded. If the **newline** option is used, it must be specified immediately after the **read** keyword, i.e., "**read newline**".
record	The **record** option causes a **read** statement to read one complete record from the keyboard. The variable list for this form of input must consist of exactly one string variable. Unused data in a previously read record, if any, is discarded. If the **record** option is used, it must be specified immediately after the **read** keyword, i.e., "**read record**".
file = *internal_filename*	For keyboard input, specify **file = stdin**. For input from files, *internal_filename* must be the name of an slx_file object (defined in a **filedef** statement).
prompt =	For keyboard input, the optional **prompt** keyword can be used to supply a prompt that is used as the title of the dialog box that pops up when a **read** from the keyboard takes place.
end =	The optional **end =** keyword can be used to supply a statement label to which SLX jumps in response to reaching the end of an input file or in response to the "End" button in the dialog box used for keyboard input.
err =	The optional **err =** keyword can be used to supply a statement label to which SLX jumps when improper input is supplied. For example, the only values that can be read into a Boolean variable are **TRUE** and **FALSE**. Any other data results in an error. In such cases, if no **err =** keyword is supplied, SLX treats input errors as fatal run-time errors..

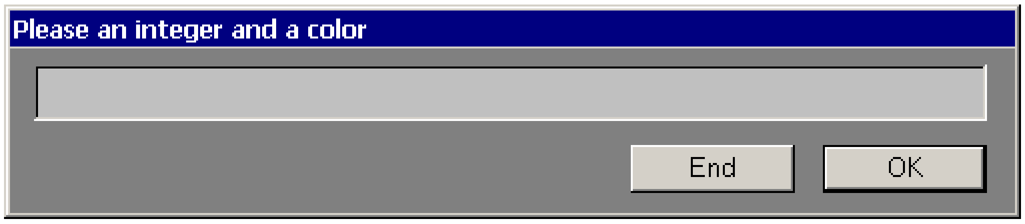
Table 2-25 – Read Statement Options

Note that the final three options shown above may be specified in any order, but they must follow **file** = *internal_filename.*

The following example illustrates reading input from the keyboard:

```
module ReadKeyboard
    {
    procedure main()
        {
        int                                     i;
        enum { red, green, blue } color;

        forever
            {
            read    file = stdin
                    end = done
                    err = trouble
                    prompt = "Please an integer and a color"(i, color);

            print (i, color)      "i = _  color = _\n";
            continue;

trouble:    print options = red,bold    "Invalid input data!\n";
            }

done:   print "EOF!\n\n";
        exit(0);
        }
    }
```

When this program is executed, the following dialog box will appear:



The following example illustrates reading input from a file:

```
module FileIO
    {
    filedef infile name = "infile.dat";

/*  11  12  13      Contents of infile.dat
    21  22  23
    31
    41  42  43
    51  52  53           */

    procedure main
        {
        int                 i1, i2, i3;
        string(100)         buffer ;

        read file=infile (i1, i2);          // read first two items of line 1
        print (i1, i2)                      "_ _\n";

        read newline file=infile (i1, i2);  // read first two items of line 2
        print (i1, i2)                      "_ _\n";

        read file=infile (i1, i2, i3);      // read from lines 2, 3, and 4
        print (i1, i2, i3)                  "_ _ _\n";

        read record file=infile (buffer);   // read all of line 5
        print (buffer)                      "_\n";
        }
    }
```

The output from the above program is as follows:

```
11 12
21 22
23 31 41
51  52  53
```

2.9.3.2 Reading from Strings

Input can be read from a string using a **read string**= statement, which has the following form:

```
    read string = string
            [ end = end_label ]
            [ err = error_label ]
            (variable,...) ;
```

Input data items are read from the text contained in the string variable specified as *string*. The **end=** and **err=** options are processed in exactly the same manner as the same options for **read file**= statements.

The following is an example of the **read string**= statement:

```
string(20)      buffer = "12345 678.9";

read string=buffer (j, x);       // Result: j = 12345, x = 678.9
```

2.9.4 Explicitly Opening and Closing Files

All the input/output examples presented thus far have relied on implicit opening that occurs with the first input / output operation. If the first operation on a file is an input operation, the file is opened for input. If the file has been defined (in a **filedef** statement) as an output file, an error occurs. The converse is true for output files.

Some applications require explicit opening and closing of files. For example, a program might write a file, close it, and reopen it for input. When explicitly opening a file, it's possible to recover from errors that would otherwise be considered fatal, e.g., trying to open a nonexistent input file.

2.9.4.1 Explicitly Opening files

The **open** statement is used to explicitly open a file. Attempting to open a file that is already open is considered an error. The open statement has the following form:

```
open   internal_filename { input | output }
       {
       name = external_filename
       options = option,...
       buffer_size = size
       window = left, right, top, bottom
       err = error_label
       }* ;
```

internal_filename must be the name of an slx_file object. Either the **input** or **output** keyword *must* be specified. The **name =**, **options =**, **buffer_size =**, and **window =** options are exactly the same as the corresponding options for the **filedef** statement. The **err =** option specifies a statement label to which SLX jumps if the file cannot be opened as specified.

The following is an example of the **open** statement:

```
    open  MyInputFile  input  name = "myfile.dat"  err = not_found ;
```

2.9.4.2 Explicitly Closing Files

The close statement is used to explicitly close an open file. The close statement has the following form:

```
close internal_filename ;
```

internal_filename must be the name of an slx_file object.

2.9.5 Determining Whether a File is Already Open

SLX provides the following macros for testing whether a file is already open:

```
file_is_open(internal_filename)
file_is_open_for_input(internal_filename)
file_is_open_for_output(internal_filename)
```

Note that each of these macros operates on an *internal* filename, i.e., an slx_file object, *not* an external filename such as "myfile.dat". The following is an example illustrates the use of the **file_is_open_for_output** macro:

```
filedef ResultsFile ...

if (! file_is_open_for_output(ResultsFile)
    {
    ...        // open the file
    }
```

2.9.6 Using a Standard Windows File Selection Dialog

The built-in procedure GetInputFileName provides an easy interface for creating a standard Windows "Open File" dialog. GetInputFileName is invoked as follows:

Boolean_variable = GetInputFileName(*FileName*, *DirectoryName*, *DefaultExtension*, *Title*);

Variable	Explanation
Boolean_variable	GetInputFileName returns **TRUE** unless a user aborts the Open File dialog.
FileName	The name of the selected file is returned in this variable. If this variable is initially non-empty, it is used as the default file name.
DirectoryName	The name of the default directory (Can be an empty string)
DefaultExtension	The default file extension to be used (Can be an empty string)
Title	The title to use for the Open File dialog (Can be an empty string)

Table 2-26 – GetInputFileName Arguments

The following is an example of the used of GetInputFileName:

```
string(100)        filename = "Input1.dat";    // selected file name  (default set to "Input1.dat")

if (! GetInputFileName(filename, "C:\\ProjectFolder", ".dat", "Please specify an input file"))
    {
    print options=red, bold "No input file selected. Execution aborted\n";

    exit(-1);
    }
```