SLX 3 Design Notes

These notes describe the features that will be included in SLX3. The new features fall into four categories:

- True local scopes.
- Changes to the sequencing of variable initialization.
- Pointers to procedures and methods **(Revised September 15, 2013)**
- The inline operator **(New description September 15, 2013)**
- An ODBC interface that permits SQL reads, writes, and queries of databases from within SLX.
- Expression-level user-defined operators (an extension of SLX's statement and macro definition mechanism).

This document describes the first four categories listed above. The latter two categories of features are largely complete, but require additional testing, examples, and documentation.

Local Scopes

Earlier versions of SLX did not provide C/C++-style hierarchical local scopes. Instead, variables defined inside locals scopes were merged into the flat scope of their containing procedure, class, module, or operator definition. Consider the following example:

```
procedure main
    {
    int   i, j;
    ...
    if (i == 1)
          {
          int       k;
          …
          k = i;
          }
    else
          {
          int       k;
          …
          k = j;
          }
```

Under earlier versions of SLX, the second definition of k was not allowed, since both definitions of k were merged into the flat scope of main. This design was implemented in the early days of SLX as an expedient that helped Wolverine get the software on-the-air more quickly. SLX assigns every variable a well-defined initial value, and the easiest way to assure that all variables were assigned their initial values was to process them all upon entry to their outer scope (in this case, main).

This approach was obviously incompatible with C/C++ and most other block-structured languages. SLX3 provides true local scopes. You can request true local scopes in two ways. In the SLX IDE, you can click Options, SLX3 and select the Local Scopes option. This sets a global default for all future SLX runs. If you wish to use local scopes in individual programs, you can use the "LOCAL_SCOPES" #ifdef symbol:

```
#define       LOCAL_SCOPES ON
#safeifdef    LOCAL_SCOPES          (Note that the "safe" form of #ifdef is preferred)
#undef        LOCAL_SCOPES
```

Enabling local scopes in the SLX IDE causes LOCAL_SCOPES to be defined. The #undef statement can be used to turn

off local scopes in individual programs when local scopes have been enabled by default in the SLX IDE.

The #define and #undef statements shown should be placed at the top of the main file of an SLX program. Local scoping rules are enforced at many points in SLX, and scoping rules must be enforced consistently throughout a program; hence LOCAL_SCOPES can be #defined or #undefined at most once in any program.

If the second definition of k in the example above is omitted, and local scopes are enabled, k will be undefined in the second scope.

Finally, note that when local scopes are enabled, upon exit from a local scope (whether by a goto statement, a return statement, or just flowing off the end of the scope), standard scope exit processing is performed in the same manner as executing from a procedure. Use counts of objects pointed to by local pointers are decremented; local sets are emptied; and use counts of local objects are decremented. If a local object has a non-zero use count after decrementing, it cannot be released. (For example, a local object may have been placed into a set.) Thus, local objects can survive beyond the execution of the local scope in which they were created. This behavior has always been true in SLX; it is inherent in SLX's use count architecture.

Strict Local Scopes

The rules for local scopes described above work well, with one fairly commonplace exception. We have encountered many user programs in which members of classes have been defined in class initial properties or actions properties, even though the intention was that such class members were to be permanent members of the class, and visible outside the class if they were public members. Strictly speaking, such definitions should be treated as local to their containing scopes; however, if SLX did this, such programs would "break" if local scopes were enabled. Hence, definitions of class members inside class initial and actions properties are treated as local only if the stronger "strict local class" rule is requested.

Strict local classes can be selected in the SLX IDE by clicking Options, SLX3, and selecting "Strict Local Classes." Within individual programs, the following can be used:

```
#define       STRICT_LOCAL_SCOPES ON
#safeifdef    STRICT_LOCAL_SCOPES
#undef        STRICT_LOCAL_SCOPES
```

As with ordinary local scopes, #define and #undef statements for STRICT_LOCAL_SCOPES should be placed at the top of the main program file and be used at most once per program.

Inline Variable Initialization

Prior to SLX3, inline initializations of variables were always collected and processed upon entry to their containing outer non-local scope. Consider the following example:

```
procedure main()
        {
        int     i = 1;

        i = 2;

        int     j = i;

#ifdef INLINE_INITIALIZATION
        print (j)       "j = _  (should be 2)\n";
#else
        print (j)       "j = _  (old SLX => should be 1)\n";
#endif
        exit(0);
        }
```

Prior to SLX3, the definitions of I and j were both processed upon entry to main. Consequently, the definition of j was processed *before* execution of the "i = 2;" statement. Hence, the printed value of j was 1.

SLX3 provides for inline initializations, in which variable declarations that include initial values are treated as executable statements, and therefore are executed in sequence.

Inline initialization can be selected in the SLX IDE by clicking Options, SLX3, and selecting "Inline Initialization." Within individual programs, the following can be used:

```
#define       INLINE_INITIALIZATION ON
#safeifdef    INLINE_INITIALIZATION
#undef        INLINE_INITIALIZATION
```

As with ordinary and strict local scopes, #define and #undef statements for INLINE_INITIALIZATION should be placed at the top of the main program file and be used at most once per program.

Because enabling inline declarations can change the behavior of previously written programs, the SLX IDE provides an option to help you track down all inline initializations. To flags all such initializations, click Options Output, Compile-Time Messages, and enable the "Warn Inline Declarations" option.

Finally, one word of caution is in order. Consider the following example:

```
for (I = 1; I <= 10; i++)
        {
        int       j = 99;

        print (j)  "j = _\n";

        ++j;
        }
```

If inline initialization is turned on, the declaration of j is treated as an executable statement, and the value printed for j each time through the loop will be 99. Because of the danger of such declarations, the SLX IDE provides an option for issuing warnings, which by default is turned on. To enable or disable such warnings, click Options, Output, Compile-Time Messages, and set the "Warn Iterated Declarations" option.

Pointers to Procedures and Methods

Prior to SLX3, pointers could only point to objects. Pointers declared as pointers to interfaces could point to any object whose class implemented the interface. Universal pointers ("pointer(*)") could point to *any* object. With the advent of pointers to interfaces in SLX2, universal pointers, which are fraught with dangers, became obsolete. In SLX3, all these features can be used as before; however, SLX3 adds pointers to procedures and methods.

Pointers to Ordinary Procedures

Pointers to procedures are declared as follows:

```
pointer(procedure (int i, double x) returning int)    p1;     // procedure selectors
pointer(procedure)                                    p2;
pointer(procedure returning double)                   p3;
```

The calling sequences of the families of procedures that can be called via a procedure pointer must be specified exactly, and only procedures with totally compatible calling sequences can be used with any given procedure pointer. "control", "inout", and "output" formal parameter prefixes must be consistent. In other words, the strictest possible compatibility is enforced.

Values are assigned to procedure pointers as follows:

```
procedure func(int i, double x) returning int
        {
```

```
        int             j;

        return 999;
        }

p1 = &func;             // assign procedure pointer value
```

Pointers to procedures are used to invoke procedures as follows:

```
i = (*p1) (i, x);       // indirect call of an ordinary (non-method) procedure
```

In the above example, a simple variable, p1, specifies the procedure to be called; however, in the general case, pointer-values expressions can be used. For example, the procedure to be called could be specified by a procedure that returns a pointer to the procedure.

Pointers to Methods

Pointers to methods are slightly more complicated than pointers to ordinary procedures, because invocations of methods specify not only a pointer to the method to be called, but also the object to which the method is to be applied. SLX3 uses the C++ "->*" operator to accomplish this. Note that the three characters comprising this operator must be typed contiguously; i.e., "-> *" specifies two operators and won't work. While a bit ugly in appearance, the "->*" operator has well-defined behavior in C++, and it serves the needed purpose in SLX3.

Pointers to methods are defined as follows:

```
pointer(method(int i, int j) returning int)     mselector1;
pointer(method returning int)                   mselector2;
```

In the previous test version of SLX3, you were required to specifiy the name of the class containing the methods, but this requirement has been eliminated.

Values are assigned to method pointers as follows:

```
mselector1= &widget::fetch1;
mselector2= &widget::fetch2;
```

Note that method names must be qualified by their containing classes.

Methods are invoked as follows:

```
i = (w ->* mselector1) (123, 456);      // Note the ->* operator
i = (w ->* mselector2) ();
```

The objects to which methods are to be applied are specified as left operands of the ->* operator, and method pointers are specified as right operands. In the above examples, instance pointers are specified as simple variables ("w ->"), Method pointers are also specified as simple variables ("mselector1" and "mselector2"). In the general case, expressions can be used both to the left and right of the ->* operator. In the following example, the pointer to the method to be invoked is contained in the object to which the method is to be applied. Both are qualified by "iw ->".

```
w2 = (iw ->* (iw -> mselector)) ();
```

If you wish to indirectly call a method from within another method of the same class, you must specify the object instance as "ME":

```
w2 = (ME ->* (iw -> mselector)) ();
```

The Distinction Between Pointers to Procedures and Pointers to Methods

SLX enforces a strong distinction between pointers to ordinary procedures and pointers to methods. The two types of pointers are *not* interchangeable. There are several reasons for this. First, the syntax for invoking procedures and methods via pointers is different. It is clear from the syntax used what type of pointer is required. For example,

```
i = (*p1) (i, x);
```

is clearly a procedure invocation, because it has no object instance qualifier. Conversely,

```
i = (w ->* mselector1) (123, 456);
```

is clearly a method invocation, as indicated by the instance qualifier and the ->* operator. The second reason for distinguishing between the two types of pointers is that it enables SLX to detect the use of the wrong type of pointer at compile time. If SLX didn't distinguish between the two types of pointers, detection of the use of the wrong type of pointer would be delayed until run time. Compile-time detection is obviously preferable.

Pointers to procedures and methods cannot be assigned to universal pointers ("pointer(*)") and conversely.

Using Typedefs

Specifying the calling sequences of complicated procedures and methods in pointer declarations can be rather onerous. If many such pointer variables are declared, using typedef statements can be quite helpful:

```
typedef pointer(procedure(double a) returning double)        FPTR;

FPTR    fptr;

procedure bind(double a, FPTR fptr) returning double

procedure joh() returning FPTR
```

The inline Operator

The SLX inline operator is a generalization of the C/C++ "test ? true_result : false_result" operator (which is also included in SLX). The following example shows how to assign x the lesser of y and z:

```
x = y<z ? y : z;
```

This is a convenient way to perform inline assignment subject to a single, simple test. The SLX inline operator also produces an inline result, but allows arbitrary computation and logic. The general form of the inline operator is as follows:

```
inline type  { body }
```

The inline *body* must contain at least one return statement that returns the value of the inline operator. The following example shows the previous C/C++ example implemented using the inline operator:

```
x = inline double { if (y<z) return y; else return z; }
```

There are no limitations on the logic or computation that can be performed in the body of the inline operator. The only rule is that all paths through the body must return a result; i.e., execution cannot fall off the end of the body.

The inline operator was deigned to be used as is in simple contexts; however, perhaps its greatest utility is use in macro expansions. The newmin.slx example shows how to implement a "min" function using a macro that expands into an inline operator.

Sample Programs

The latest test version of SLX3 includes the following sample programs:

AdvancedFptrs.slx	An example of advanced use of pointers to procedures.
CPPscopes.slx	Comprehensive example of local scopes – worth stepping through carefully. Be sure to select Monitor, All Objects.
Earlyexit.slx	Comprehensive example of what happens on exit from local scopes.
InlineInit0.slx	Simple example of inline initialization.
JumpIntoIteration.slx	An example of the evils of jumpting into iterations.
Localscopes0.slx	Simple example of local scopes.
Newmin.slx	An example of how to implement a “min” function as a macro that expands into an inline operator.
PointersToProcs	A simple example of pointers to procedures
Strictlocalscopes0.slx	Simple example of strict local scopes.
TripleFptr.slx	An example of very complicated indirection using pointers to procedures. This is not a style to be emulated, but it is an interesting example of what can be done *in extremis*.
TypedefFptrs	An example of using typedefs with procedure pointers.
Xfaceptrtoprocs.slx	An example of interfaces specifying pointers to methods.

All of these examples are worth looking at *very* carefully. Use the SLX Debugger to step through them. Be sure to enable the Monitor All Objects display.