

The SLX IDE

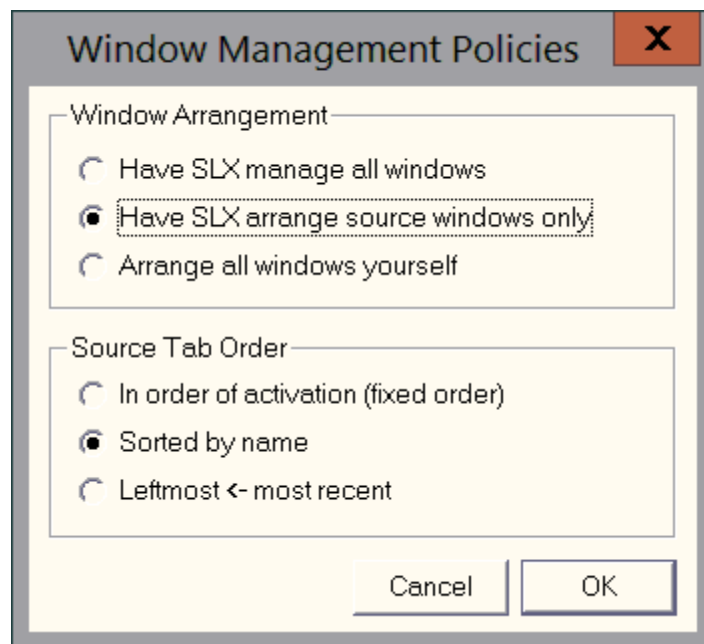
1.0 Introduction



This memo provides a quick overview of the SLX interactive development environment (IDE). The IDE contains a built-in source viewer/editor that is based on Microsoft's Rich Edit feature. If you are familiar with Microsoft Word, WordPad, or NotePad, the editing paradigm will be familiar to you. Every effort has been made to exploit built-in Windows features in standard ways.

2.0 Screen Layout

The IDE screen is divided into six areas, from top to bottom: (1) the window caption/menu bars, (2) a toolbar, (3) the Log and Calls & Expansions windows, (4) a source code window, (5) a tab bar for selecting source windows, and (6) the status bar. When the IDE is invoked, the default split between the log window and the editing window is 20% for the log and 80% for editing. This division can be altered by dragging the border between the two areas. The log window can be minimized, but it cannot be closed. (If we let you close the log window, we'd have no reliable place in which to post important messages.) Clicking on the Log window's "X" button clears the window.

The IDE's windows can be managed in three ways: (1) you can have SLX manage the placement of all windows; (2) You can have SLX manage the placement of source windows only and manage non-source windows yourself; or (3) you can manage the placement of all windows yourself. When SLX manages source code windows, it uses a single large window to show one source file at a time. The active source window can be changed by clicking on source window tabs near the bottom of the screen. When you manage source windows yourself, you can show multiple source windows at the same time. Source tabs can be sorted in three ways: (1) in order of activation, (2) in alphabetical order, or (3) in activation order (leftmost = most recently activated). You can select window management options by clicking Window, Window Management Policies:



If the screen becomes messy, you can have SLX rearrange the screen by clicking on the  button on the toolbar. To compile the program, you need to make the window containing the main program the active window, if it is not already the active window, by clicking in it. (Under Windows, the active window is always highlighted.) When you click on the  button, compilation will start with the active window. If the active window is not the window containing your main program, only a portion of your program will be compiled (the file whose window is active and any files it imports), and you will see the infamous "No Main Program" message in the log window.

If errors are encountered, they are highlighted in red in the appropriate source windows. If an error occurs in an imported file for which no source window has been opened, a source window is automatically created. If some

windows contain errors and others do not, windows containing errors are given priority in SLX's window management scheme.

3.0 Font Selection

SE allows you to choose a typeface and point size. The selected typeface is used in all source windows and for output to the screen and to Rich Text Format (RTF) files. RTF files are described in section 5.0. You should choose a proportional font. In the universe of fonts, we have found only two readily available fonts that work well, Arial and Microsoft Sans Serif. However, if you want to use Times Roman or some other font, you may do so. You will probably want to use either 10-point or 11-point type. Type that is smaller than 10 points is too small for most people to read, and type that is larger than 11 points results in output that is too wide to view or print. On occasion, we use 12-point type when demonstrating simple programs.

4.0 Use of Colors and Boldface Type

When you compile a program, SLX will stylize visible source windows, using the following scheme:

<u>Color</u>	<u>Style</u>	<u>Significance</u>
Dark Blue	Normal	Normal source code
Dark Blue	Boldface	Indicates an SLX reserved word, e.g., "if"
Bright Blue	Boldface	Indicates a macro defined statement name (built-in or user-defined)
Purple	Normal	Indicates an SLX built-in function
Red	All	Highlights errors at their points of occurrence
Dark Gray	Normal	Comments

If you click on text with your right mouse button, a pop-up menu will appear, presenting you with several options. If the text you click on is bright blue, indicating a macro or defined statement, the pop-up menu will give you the option to display the expansion of the bright blue text. If you right click on the bright blue text again, you will be given the option to hide the expansion. If you click on the text of an executable statement after your program has been compiled, the pop-up menu will give you the option to set/clear a breakpoint. Additional options allow you to get descriptions of variables, examine their values, specify that they should be "watched" for changes, etc.

5.0 The Use of .RTF Files

SLX makes extensive use of Rich Text Format (RTF) files. Internally, SE's editor is RTF-based. RTF allows the use of stylized proportional font text. Many Windows components are RTF-aware. For example, Microsoft Word can read and write (Version 6 and above) RTF files. The clipboard is RTF-aware. If you're preparing a report, you can copy or cut information from SE screens and paste it into a Word document.

If you're sending output to a file that has an ".RTF" extension, the file will be written in RTF format. You can also explicitly request RTF for a file that has an extension other than ".rtf". All other file output is written as plain ASCII text.

6.0 A Quick Tour of SE's Menus

6.1 The File Pop-Up Menu

SE's File menu is pretty much what you'd expect of a Windows application. Menu items are available for opening, saving, closing, printing files, etc. Because RTF files are so heavily used by SE, an "Open RTF" item has been placed in the File menu. A menu item is provided for clearing the log window. This can be convenient if you are making multiple runs and want to "clear the slate."

6.2 The Edit Pop-Up Menu

SE's Edit menu is again pretty much what you'd expect of a Windows application. A few non-standard items have been added for your convenience. For example, F3 is widely used as a "Find Next" keyboard accelerator. We have taken this a step further and added "Shift F3" for "Find Previous." If you've spent much time using Microsoft editors, you know how frustrating the lack of a "Find Previous" function can be. For files containing errors (compile-time or run-time), we have added menu items that allow you to jump ahead to the next error or jump back to the previous error. "F4" and "Shift F4" keyboard accelerators are available for these items, as well as icons on the toolbar.

6.3 The Options Pop-Up Menu

The Options menu contains eight items.

1. Clicking on SLX2 toggles the SLX2 feature set on and off. This item is provided for compatibility with old SLX1 applications. The SLX2 feature set introduced object-oriented capabilities to SLX. (Documentation is available in the Wolverine\SLX\doc\SLX2 folder.) New users should always enable the SLX2 feature set. If SLX2 features are disabled, the compiler will fail to recognize object-oriented keywords such as "method."
2. Clicking on SLX3 brings up a popup menu that allows enabling/disabling of SLX3 features. (Documentation is available in the Wolverine\SLX\doc\SLX3 folder.)
3. Clicking on Editor/Display brings up a popup menu that allows you to configure SLX's source editor.
4. Clicking IDE allows you to configure the behavior of the SLX IDE..
5. Clicking on Output allows you to select the content of SLX's output, the destination to which it is written, and whether or not specified families of error messages are enabled..
6. Clicking on Browser/Debugger allows you to configure the behavior of SLX's source code browser and debugger.
7. Clicking on Execution allows you to specify how SLX programs are executed. The only option that most users should use is the "Run at Background Priority" option, which is useful for executing long-running simulations, while continuing to work with other windows. Simulation applications tend to consume 100% of the CPU.
8. Clicking on Security Key brings up a popup menu that allows you to specify how Wolverine security keys (dongles) are handled. For example, you can specify that your key is located on a network server. This option is provided only in commercial versions of SLX. Student SLX does not require the use of a dongle.

6.4 The Window Pop-Up Menu

Clicking on Window reveals a popup menu for manipulating SLX windows. Most of these options are self-explanatory; however, SLX's management of windows deserves an explanation. You can choose three window management policies: (1) have SLX arrange all source windows automatically, (2) have SLX arrange source windows only, or (3) manage all windows yourself. When debugging a large SLX program, it's common to have many open windows. (Using a dozen windows is not uncommon.) Therefore, it's usually convenient to have SLX manage windows automatically. SLX's automatic management places high-priority windows first, lower priority, but non-source windows second, and source windows third.

In addition to specifying a window management policy, you can specify your own custom window sizes. For example, you may prefer to use a very large log window. Once you've tweaked the sizes of all windows on the screen, you can save their sizes as defaults to be used henceforth.

6.5 Compile

Clicking on *Compile* will do the obvious.

6.6 Run

Clicking on *Run* will run or continue execution of a program. You can compile and run a program in two separate steps, or you can click on Run, and a compilation will be done automatically. A program can be run only if it contains no compilation errors.

6.7 The Monitor Pop-Up Menu

6.7.1 Puck-Based Displays

The Monitor menu allows you to enable up to five debugging windows that portray the status of pucks in a program. The “Moving Pucks” window displays pucks that are currently eligible to move through a program (the Current Events Chain, in GPSS/H parlance). The puck that is *currently moving* is also shown in the status bar at the bottom of the screen. The “Scheduled Pucks” displays pucks undergoing scheduled time delays. The “Waiting Pucks” window displays pucks that are delayed in “wait until” statements (waiting for one or more program components to reach a given state) or in “wait” statements (no “until” clause given), sleeping until awakened by another puck. The “Interrupted Pucks” window shows pucks that have been interrupted by other pucks. Finally, the “All Pucks” window shows all pucks in a single window.

The pucks in a puck display window initially appear in their “natural” order. For example, pucks in the Scheduled Pucks window are shown in ascending time order. You can alter the sorting by clicking on the appropriate column heading. For example, if you click on the “Object Class” heading, the pucks will be sorted by name. If you click on Object Class again, the pucks will be sorted by descending name. Sorting makes it easier for you to find pucks for which you are looking.

If you click on the class name of a puck with the left mouse button, source code windows will be updated to show the state of the selected puck. If you click with the right mouse button, you will be given the options of displaying the puck itself, displaying the puck’s object, and setting/clearing a “trap” for the specified puck. When the trap is set for a given puck, each time it attempts to move through the program and each time it can move no further, you will be notified.

6.7.2 “All Objects” Display

Clicking on All Objects in the Monitor menu will create a window showing *all* objects. By default, objects are displayed sorted by ascending object identifiers, e.g., “myobject 21”. Different sorting can be chosen by clicking on the window’s column headings. Objects whose instances have been declared, the name of the variable used in the declaration is shown in the “Name” column. For objects that are dynamically created, the Name column is blank. Consider, for example, a declaration of the form “widget w1, w2;”. The widget named w1 might be “widget 21”, in which case the widget named w2 would be “widget 22”. The numbers 21 and 22 simply indicate that w1 and w2 are the 21st and 22nd widgets to be created.

The All Objects window is automatically updated as new objects are created and old ones are destroyed.

If you would like to display a given object, but you don’t know how to select it, you can use the “Find” menu item, described below.

6.7.3 Global Data Display

Clicking on Global Data in the Monitor menu will create a window showing all global data. Data initially appears sorted by module, however, by clicking on column headings, you can change the display order.

You can right click to get further information about variables displayed in this window. When you right click on a variable name, a pop-up menu will appear, giving you a variety of options. For all variables, you will be given the option of displaying the variable’s definition. When arrays are displayed, “[array]” is shown as a variable’s “value.” For arrays, the pop-menu gives you the option of displaying the array. For sets, “[set]” is shown as a variable’s “value.” For sets, the pop-up menu gives you the option of displaying the set’s members and issuing a report for the set. All sets have a report property. A set’s report property issues a report request for each member of the set. For pointer variables and object instances, the pop-up menu gives you the option of displaying an object and opening a report window if the object has a report property.

When an object is displayed, the object’s window shows elements of the object. If you right-click on an object’s element name, a pop-up menu will appear, presenting you with the same options described in the preceding paragraph. If you right-click on an object’s use count (always displayed) you will be given the option of displaying all uses of the object. This can be very useful for debugging.

Arrays are displayed in a vertical format. Each of the indices, e.g., row, column, and higher dimensions, has its own column in the display. By default, data is shown in ascending order, with the highest index varying most rapidly; however, you can click on any column heading to display the array sorted by that column. If you click a second time on a given heading, the data is displayed in reverse order, sorted on that column. For example, if you want to view an array by column, rather than by row, you can simply click on the “Column” column heading.

6.7.4 Local Data (Static Selection) Display



Clicking on Local Data (Static Selection) in the Monitor menu will create a window showing local data for the active puck at its current level of call. The created window is “sticky,” i.e., it always displays local data for the puck and level of call that existed when the window was created. If the local data “goes away,” the designation “(Destroyed)” will be placed in the window’s title. This can happen, for example, to local data for a procedure when the procedure returns to its caller. Data initially appears in the order in which you defined it in your program; however, by clicking on column headings, you can change the display order. The right-click options described for the Global Data window are also available for Local Data windows.






6.7.5 Local Data (Dynamic Selection)

Clicking on Local Data (Dynamic Selection) in the Monitor menu will create a window that always shows local data for the currently active puck at its current level of call. The window is quite dynamic. When a given puck calls or returns from a procedure, the local data window is automatically updated. When a change of pucks occurs, either as a consequence of program execution or at your request (via mouse clicks requesting puck-based information), the local data window is also updated. The window title for a dynamically selected local data window always contains the designation “Current.”

6.7.6 Data Display Example

The following example will lead you through many of SLX’s data display features. It will take you about ten minutes to work your way through this example, but it will be well worth your time. Start by running SLX in full-screen mode and selecting case1h.slx (in the Wolverine\SLX\Sample folder) as a test program.

- A. Click on the check mark icon to compile the program.
- B. Scroll the source code down to about the $\frac{3}{4}$ point.
- C. Right click on the statement “remove job from printer_queue” and set a breakpoint on it.
- D. Click the  icon to run up to the breakpoint.
- E. Click on the Monitor menu item and select “Global Data”.
- F. Click on the Monitor menu item and select “Local Data (Dynamic Selection)”.
- G. If necessary, drag the bottom of the Local Data window down until all local variables of printer 1 (the current and only instance of a “printer” object) are visible.
- H. Right click on “job” in the Local Data window, to display the object pointed to by “job”. (Click on “Show *job”.) Note that “job” was set to point to the first object in the set “printer_queue” in the preceding statement.
- I. Note that the use count for “printer_job 1” is 3. Two of the three uses are obvious. First, we know that printer’s “job” variable points to printer_job 1. Second, we know that printer job 1 is in the printer_queue set. (This counts as a use.) What is the third use? Right click on printer_job 1’s use_count and click on “Show printer_job 1’s Uses (3)”.
- J. The first use shown indicates that there is another variable named “job” in main that also points to printer_job 1. The second and third uses are the obvious ones discussed above. Why does this variable point to printer_job 1? We’ll find out in a minute, but first let’s display some additional information and carefully step through the program.
- K. Right click on “printer_queue” in the Global Data window, and click on “Show printer_queue 1’s Members”. This will corroborate what we already know: printer_job 1 is in the printer_queue set.
- L. Click on the  icon in the toolbar, executing the remove statement. Note that printer_job 1 disappears from the printer_queue set window, the use count of printer_job 1 is decremented from 3 to 2, and the “printer_queue” usage of printer_job 1 disappears from printer_job 1’s usage window.

- M. Click on the  icon repeatedly until you are poised to execute the “destroy job” statement. Click once more on the  icon. Note that the printer_job 1’s use count is decremented from 2 to 1, and the “job” usage of printer_job 1 disappears from printer_job 1’s usage window,
- N. Click on the  (“get next puck”) icon in the toolbar. The next puck to execute is main 1/3. Main 1/3 is used to model the arrival of priority 2 (worker) jobs into the system. Note that the dynamically selected Local Data window is automatically updated to show local variables of main 1.
- O. Click on the  toolbar icon to execute the “job = new printer_job” statement. Assigning a new value to “job” causes the use count for printer_job 1 (the object that “job” previously pointed to) to be decremented from 1 to zero. When printer_job 1’s use count goes to zero, it is automatically destroyed, and a “(Destroyed)” designation is added to the title bar of printer_job 1’s local data and usage windows.
- P. Kill the printer_job 1 windows, and repeatedly click on the  toolbar icon until printer_job 42 shows up in the printer_queue set window.
- Q. At this point in the simulation, some very interesting queueing takes place in the model. Job priorities come into play. If you take about 5 minutes to very carefully step through the mode and display appropriate data windows, you can verify that the model works as it’s supposed to, and you’ll experience the power of SLX’s debugging features.

6.8 The SLX Browser

Clicking on Browse opens an SLX browser window. Browser windows are similar to Windows Explorer-style windows, with a tree-view shown to the left and textual or tabular information shown to the right. The tree-view contains subtree nodes, which can be further expanded, and leaf nodes, which cannot. Subtree nodes are drawn with square icons containing a plus sign if the node is *not* currently expanded, and a minus sign if the node *is* currently expanded. Clicking on a subtree node with the left mouse button toggles its status with each click. Leaf nodes of the tree-view are shown with circular icons. Clicking on a leaf node causes a display to be shown in the right portion of the browser window. Only one leaf node can be displayed at a time. If you want to show more than one leaf node at a time, you must open multiple browser windows.) Clicking on a leaf node with the left mouse button reveals a corresponding display.

By default, SLX-defined (“system”) components are excluded from browser windows. However, by clicking on the Options menu item and then clicking on Browser/Debugger, you can specify that you’d like to have system components included. If you’re interested in exploring details of SLX itself, you should enable the inclusion of system components in browser windows. If you wish to focus on code that you have written, disabling the display of system components will significantly reduce the size of browser displays. In the sections that follow, the tree-view items provided are briefly described.

6.8.1 Flat Source

Clicking on Flat Source reveals six subtree nodes: Modules, Classes, Procedures, Variables, Statements/Macros, and Typedefs. Clicking on any of these subtree nodes will reveal an alphabetized list of all components of the specified type in the source code comprising the current compilation. For example, clicking on Procedures will show the names of all procedures in the current compilation. If you click on any of the names, source code will be shown in the right side of the browser window. If you are searching for a program component and don’t have a good idea of where it might be located, the Flat Source tree-view node provides a quick way to search.

6.8.2 Hierarchical Source

Clicking on Hierarchical Source reveals an alphabetized list of the files comprising the current compilation. Clicking on a file name reveals an alphabetized list of modules defined within the file. Clicking on a module name reveals subtree nodes for Classes, Procedures, Statements/Macros, and Typedefs defined within the module. Clicking on any of these nodes reveals an alphabetized list of components of the specified type. If you need to examine a program component, and you know exactly where to look for it, the Hierarchical Source tree-view node provides a quick way to home in on the component. If you click on a component that contains variables, a node is added to the browser tree, allowing you to display an alphabetized list of the variables.

6.8.3 Objects

Clicking on objects reveals an alphabetized list of classes defined in the current compilation. Clicking on any of the displayed names will display all objects (both statically and dynamically created) in the right portion of the browser window. If you step through your program when such a display is active, the display is automatically updated as new objects are created and old ones are destroyed.

6.8.4 Pucks

Clicking on Pucks reveals three subtree nodes: Moving, Scheduled, and Waiting. Clicking on any of these displays all pucks in the specified state in the right portion of the browser window. If you continue to step through your program, this display is updated as new pucks are created, old ones are destroyed, and displayed pucks change state.

6.9 The Step Pop-Up Menu

The SLX debugger provides a number of powerful options for stepping through a program. Each of the options in the Step menu can be invoked using keyboard accelerators (F7 through F12) and by using icons on the toolbar.

“Step Next” executes your program until a new puck is picked up. “Step Drop” executes until the current puck is dropped, i.e., until it can move no further through the program. “Step N” executes the next N statements of your program. Note that for Step N, all statements, visible and invisible, are counted. Invisible statements may be concealed inside macro/statement expansions and/or inside lower-level subroutines. “Step Over” executes one visible statement. It steps over macro/statement expansions and procedure calls. “Step Into” executes one statement. If the executed statement is hidden inside a macro/statement expansion, the expansion of the macro/statement is revealed. If the executed statement is a procedure call, the source window is updated to show the source code for the procedure. If Step Into steps into source code in a file for which no window has previously been opened, a window is automatically opened. “Step Out Of” steps out of the current procedure. If Step Out Of is used when you’re already at the highest level, it is treated as Step Over.

6.9 Find

Clicking on *Trap* activates a dialog box that gives you the option of trapping some special conditions. You can specify a time trap. When program execution reaches the specified time, you will be notified.

6.10 Trap

Clicking on *Trap* activates a dialog box that gives you the option of trapping some special conditions. You can specify a time trap. When program execution reaches the specified time, you will be notified.



You can request notification each time a new puck is about to move through your program and/or each time the current puck can move no further. These traps are useful for “microscopic” debugging. By using them, you can assure that any given puck you are tracking through a program will not “escape” without your knowing it. In contrast, when you use a “step” command and the puck you are following is blocked at its next statement, new pucks will be tried until one of them *can* execute a single statement. Potentially many pucks can be examined without your knowledge.

You can request that individual pucks be trapped before these pucks have been created. Such requests are enqueued and honored when the specified pucks first enter a program. For example, if you know that the puck named “load 273/1” caused an unusual problem in a previous run, you can set a trap in advance of load 273/1’s creation, assuring that you can track its entire life history, if necessary, to find the problem.

6.11 The Help Pop-Up Menu

Clicking on *Help* reveals a menu that allows you to (1) Get online help (“Procedural SLX”, (2) show information “About SLX”, (3) display all unreferenced variables in a program, (4) show a summary of memory usage, (5) show a detailed memory usage report, (6) show free memory details, and (7) show properties of your computer’s CPU.

7.0 The IDE Toolbar

The IDE toolbar contains convenient, single-click shortcuts for actions that would otherwise require multiple clicks using menus. If you hold the mouse over a toolbar item for a second or so, “tooltip” text will be shown, explaining the function of the toolbar item. The  (“run”) tool changes to a  while a program is running. Clicking on the stop sign will stop program and allow you to explore the program, using the debugger.

8.0 The Status Bar

During program execution, the status bar at the bottom of the screen shows (1) the current simulated time, (2) the current puck, its status, and its priority, (3) the CPU time consumed by the current run, (4) the amount of memory currently in use, and (5) the current line of the active window.

9.0 The Calls & Expansions Window

The Calls & Expansions window is the single most powerful debugging tool of SLX. It is a compact tool for quickly exploring (via left mouse clicks) source code and data. The Calls & Expansions window is located to the right of the Log window. The Calls & Expansions window uses a Microsoft Windows “Tree-view” to depict the current state of your program. Each node in the tree represents an active “level” of your program. Each node in the tree comprises a graphical symbol and the name of an object, procedure, macro, or defined statement. Note that this implies that there are two different kinds of nodes in the tree, procedure and object nodes, and nodes representing “soft-wired,” expandable definitions.

The following notations are used in graphical symbols that annotate the tree:

- Shape** The program component that is executing is shown with a round-shaped symbol, evocative of a puck. Since the executing component is always at the bottom of the tree, this notation is not strictly necessary, but I find it helpful. Higher-level procedures, macros, and statements are shown with square-shaped symbols.
- Color** The executing component is shown in red, and higher level components are shown in green. Once again, the distinction is not strictly necessary, but I find it helpful. If you suffer from red/green color blindness – 11% of all men do – you won’t be missing much. The same colors are used in the tree as are used to highlight source code in source code windows.
- + and -** Macros and defined statements are always depicted with a plus sign or a minus sign in their symbols. Clicking on a plus sign causes the expansion of the macro/statement to be revealed, and clicking on a minus sign causes a visible expansion to be hidden. (This is a familiar convention in Windows.) Note that you can also control the visibility of expansions by right-clicking on bright blue macro/statement names and clicking the desired option in the pop-up menu that appears. Using the tree-view is easier.
- Fill** A filled-in (solid) symbol is used to annotate the component that is shown in the source window, vertically centered, underlined, and highlighted in the same color as the symbol in the tree-view. Hollow symbols are shown for components that are in the active call tree, but not currently selected.

It’s well worth your time learning how to use the Calls & Expansions window. For starters, you might want to try the following experiment with `barb137.slx`:

- A. Click on the “Step Over” icon several times, until the seize statement is highlighted.
- B. Click on the “Step Into” icon twice, once to reveal the expansion of seize and once to step into the SEIZE procedure that implements the seize statement.
- C. Scroll the source code upward to a point at which you can see the report property for the facility class.
- D. Right click on the first print statement inside the report property, and set a breakpoint on it.
- E. Click on the “Run” icon, and the program will execute until it reaches the breakpoint.
- F. Drag the bottom of the Calls & Expansions window down until you can see its entire contents. At this point, you will see a tree that is eight levels deep. The solid red circle in the next-to-last level tells you that code is being executed inside facility named joe. The last level of the tree has a red circle with a plus inside it, for the print statement on which you set a breakpoint.

- G. Click on the plus sign to reveal the expansion of the print statement, and click on the minus sign that subsequently appears, to once again hide the expansion.
- H. Click on the Monitor pop-up menu item, and select “Local Data (Dynamic Selection)”. At this point, a local data window will appear, showing the data for facility joe.
- I. Click on the hollow green square in front of main 1/1, at the top of the tree, and several things will happen. The source code for main will be highlighted in green and centered in the source window. The square in front of main in the tree-view will be filled in to solid green, indicating that main’s source code is selected. Finally, the local data window will be updated to show main’s data. Unfortunately, for purposes of this exercise, it has none.
- J. Note that main is executing a “report(system)” statement. What is system? Since it’s not a local variable of main, it must be global. Click on the Monitor pop-up menu, and select Global Data. Click on the “Variable” column header to sort by variable name, and scroll the list down until you see “system.” You will see that system is a “system_reporter” object. If you look at the second level of the tree, this is exactly what you will see.
- K. Click on the “system_reporter system” level of the tree to reveal what the system_reporter object does. It prints a heading and issues a “report(entity_reporters)” statement. What kind of a variable is entity_reporters? It is a set that is a local variable of the system_reporter object. You can see entity_reporters in the local data window, and you can see its declaration in the system_reporter object, by scrolling up the source code.
- L. What happens when a report is issued for a set? To see, click on the next lower level in the tree. When you do so, you’ll be positioned inside the definition of a set in system.slx. The report property for a set includes a “for each” loop that issues a report for each member of the set. The pointer variable “ptr” is used to perform the iteration. “*ptr” denotes the object to which “ptr” points. What kind of an object is ptr pointing to at this time? It’s pointing to a “facility_reporter” object.
- M. The next lower entry in the tree (“entity_class”) has a plus sign in it, indicating that it represents a macro or defined statement. Skip over it for the time being and click on the following entry (“facility_reporter_class facility_reporter”). When you do so, you won’t see much, because the definition of the facility_reporter object is hidden inside the “entity_class” statement.
- N. Click on the “entity_class” tree entry, and the expansion of the entity_class statement will be revealed.
- O. Note that the facility_reporter is executing a “report(facility_set)” statement. What is facility_set? You can see it in the expansion of the entity_class statement. You will note that it is defined *outside* the facility_reporter_class definition. It is, therefore, not local, but global. If you look through the global data window, you will find it. It is a set of facility objects.
- P. Click on the “set facility_set” tree entry, and you will see how the report property for this set is processed. (This is your second look into the definition of a set in system.slx.) This time, “ptr” points to facility joe.
- Q. Finally, click on facility joe, and you’ll be back where you started when the breakpoint was reached, only now (with perhaps some additional review), you’ll know how you got there.

10.0 Summary

You will find that the SLX user interface is easily mastered. Given the complex nature of man-machine interaction, you may have some suggestions for us on how SE’s user interface could be improved. While we can’t tailor the user interface to the whims of a particular user, we can and will incorporate changes that, by consensus, improve the product. Welcome to simulation under Windows, and have fun!