Sample SLX Programs

This memo describes sample programs distributed with SLX. We'd strongly suggest that you examine all of these programs very carefully. If you wish to get hard copy listings of the source code, open the source file, click on the checkmark icon on the toolbar to compile the program, and then click on the printer icon on the toolbar. The SLX compiler will highlight SLX reserved words, e.g., "if," "for," etc., making the source code easier to read. The test programs described in this memo are stored in the Wolverine\SLX\Sample folder. Most, but not all of the examples can be run with the student version of SLX.

The following files are included. They are shown in recommended viewing order.

Program	Description
CASE1A CASE1B CASE1C CASE1D CASE1E CASE1F CASE1G CASE1H	This group of models is described in detail in the "Getting Started with SLX" memo.
BARB138	This program is a one-line, single-server queueing model constructed from higher-level, GPSS/H-style building blocks written in SLX.
H7	This file contains definitions that implement a subset of GPSS/H in SLX. It is imported into a number of the sample programs. H7 is located in the Wolverine\SLX folder.
H8	H8 is a version of H7 that was updated to take advantage of SLX2 and SLX3 object-oriented features. H8 is located in the Wolverine\SLX folder.
STATS	This file contains the definitions of SLX's statistical features. It is located in the Wolverine\SLX folder.
SUBSCERR	This program illustrates SLX's run-time diagnostics. It purposely generates a subscript value that is out-of-range. Note how the point of occurrence and the offending value are indicated.
DIAGRUNT	This program illustrates the use of SLX's "diagnose" statement. A subroutine examines its argument and issues a run-time error if it is out-of-range. Two things are noteworthy about this example. First, note that the diagnostic message is placed at the point of invocation of the subroutine, showing exactly where the bad argument value came from. Second, note that the "diagnose" statement makes available to SLX users the same diagnostic message facility that SLX itself uses.
TECH1M	This is a model of a "reentrant" machining line. Parts are processed in three stages. Stage 1 is performed by machine 1. Stage 2 is performed by machine 2, and Stage 3 is performed by machine 1 (revisitation). What makes this model interesting is the implementation of deciding what operation should be performed next, based on queue contents.
TECH2M	This is the same as TECH1M, but written using an active-server world-view.
RENEGE2A	This model illustrates the use of SLX's fork primitive to model simultaneously waiting for availability of a server or for a maximum waiting time to elapse, whichever comes first.
RENEGE2B	This model uses a compound wait until containing a time-based comparison to accomplish the same results as RENEGE2A. This model was built after we made improvements to wait until to allow time-based comparisons.
CALLCENTER	This is a sample model of a call center.

QUICKLINE	This is a model of a bank that employs a single queue for multiple tellers.
SLOWLINE	This is a model of a bank that uses N queues for N tellers.
BUSPARADOX	This model illustrates a result from renewal theory. Suppose that buses are supposed to arrive at a bus stop every 12 minutes. If you walk up to a bus stop at any arbitrary random time, you would probably expect to wait 6 minutes for the next bus to arrive. This situation is more complicated than you would think. The expected wait time depends not only on the mean of the bus interarrival time distribution, but also on the variance. For bus IAT distributions other than exponential, the expected waiting time is greater than half the mean.
PICKBEST	This program models an interesting conveyor system; it is described in detail in SLX97.PDF The program generates a trace file, PICKBEST.ATF, which can be run with PICKBEST.LAY to produce a nice animation of the conveyor system.
PICKBST2	This is a "smart loads" variation of PICKBEST.SLX.
PBESTDLL	This is a variation of PICKBST2.SLX that runs concurrently with the DLL version of P5 (2D), rather than generating a trace file for post-processed animation.
PBEST3D	This is a variation of PICKBST2.SLX that runs concurrently with the DLL version of P3D (3D), rather than generating a trace file for post-processed animation.
CONVEYOR	This file contains source code for the conveyor modeling package that is used by PICKBEST.SLX and PICKBST2.SLX.
CVRSTMAC	This file contains source code for macros and statement definitions used by PICKBEST.SLX, PICKBST2.SLX, and CONVEYOR.SLX.
PROOF4	This file, located in the Wolverine\SLX folder, contains the SLX-P5 interface. It is imported by PICKBEST, PICKBST2, and PBESTDLL. This file is worth studying even if you don't intend to use Proof Animation with SLX, as it contains a wealth of examples of how to use SLX user-defined statements. For every P5 trace file command, there is a corresponding SLX statement definition to format the command and write it to a trace file. The one-to-one relationship makes it easy to animate an SLX model.
PROOF3D	This file, located in the Wolverine\SLX folder, contains the SLX-P3D interface. It is imported by PBEST3D.
NEWSTATX BARBANTI BARBCOMN BARBBM TJSLOCK2	This group of models illustrates SLX's statistical features. The models are described in detail in the "SLX Statistical Features" memo.
CALC	This program is a primitive desk calculator program. It reads expressions, parses them using recursive descent, and prints their values.
SETS3 SETS4	These programs illustrate SLX's built-in *set* capabilities.
SETMEMBR	This program illustrates the use of built-in operators for testing whether a specified object belongs to a specified set.
PHONBOOK	This example illustrates the use of SLX sets to record entries in a simple phone book.
BIGRANK2	This program illustrates the efficiency of SLX's ranked sets. A generalized version of Henriksen's algorithm for the future event set has been implemented for SLX ranked sets. Time-consuming linear searches are avoided.

BIG-MANUAL-RERANK	SLX allows sets to be ranked by ascending and descending values of a specified collection of attributes of the objects placed in the sets. Changing the value of a ranking attribute while an object is in a ranked set is a no-no. If you want to change the value of a ranking attribute of an object, you have to remove the object from its set, change the value of the ranking attribute, and reinsert the object into its set. This program illustrates a manual approach for doing so.
BIGAUTO-RERANK	SLX also supports automatic reranking. (See the above example.) For autoreranked sets, the ranking attributes must be declared as control variables. The SLX compiler generates code to detect changes in control variables, and if a control variable is used as a ranking attribute for one or more sets, to automatically perform reranking .
ENUMFLPS	This program illustrates the use of enumerated types (enums). If you've never used enums, you should study this program carefully. Enums are frequently the vehicle of choice for assigning symbolic names to states that an entity can be in. For example, "idle", "busy," and "breakdown" are more meaningful than "1,", "2," and "3" as state designators.
ENUMITER	This program illustrates enum-based iteration (very handy).
WEEKDAYS	This example illustrates the use of enumerated types as array bounds and enum values as array subscripts..
JOHSTRNG	This program illustrates SLX's string manipulation capabilities.
BIFTEST	This program illustrates a number of SLX's built-in math/trig functions.
READALL	This program illustrates SLX's input capabilities.
RWSTEST	This program illustrates the string= option of read/write. Instead of reading/writing a file, one can read/write a string variable. This provides a way of doing data conversions as pseudo-I/O operations. For example, one could "read" a string containing an ASCII representation of a floating point number, placing the result in a floating point variable.
YIELDTO	This program illustrates the use of SLX's *yield to* statement. This feature allows users to assume control over deciding which puck gets to execute next. When yield to is used, the active puck immediately yields control to a specified puck.
BARBRIV	This program illustrates the use of SLX's *random_input* statement. This statement makes it easy to get summary statistics for the random variates that are actually generated from a specified input distribution. Options are also provided for truncating either tail of a specified distribution.
BARBREAL	This program is a real-time version of a GPSS/H-style barbershop model. One minute of simulated time is scaled to 1/10 second of real time. The simple technique illustrated in this model can be exploited to turn SLX into a real-time language.
PUCKSTATES	This program illustrates the use of a puck's *state* variable. At any given time, a puck can be MOVING, SCHEDULED, WAITING, INTERRUPTED or TERMINATED. This program illustrates how each of these possibilities can arise.
BITOPS	This program illustrates C-style bit operations available in SLX.
PINGPONG	This program illustrates the speed of wait until. A parent puck and its offspring execute 1,000,000 alternating wait untils. On a fast machine, this takes around two seconds.
NEWXERR	This program illustrates a number of execution errors that can occur in puck-based scheduling operations. Several different failure modes are illustrated. All but one are "commented out." By selectively commenting out source code, you can explore all the possibilities;
DET?ARAY	These programs illustrate some of the complexities of object use counts. The point of these programs is to illustrate how objects are released only after their use counts have gone to zero. Use counts can be reduced by pointer assignments, explicit *destroy* operations, by having pointer

variables local to a procedure that returns, or by having pointer variables inside an object that itself is destroyed.

AUGMENT This program shows how prior definitions for SLX objects can be extended.

COMPTIME A *precursor* module is compiled in its entirety when the "}" ending its scope is encountered. Thus, definitions inside a precursor module are available at compile time. This example contains a precursor module that defines a file, a set, and two statement definitions. The first statement definition reads from the file (at compile time!) and creates objects and places them into a ranked set. The second statement definition removes the first element from the set each time it is called, and generates an executable statement. If the set is empty, a statement that reflects this condition is generated

DEFAULTS This program shows how to establish default values for statement operands. To see how it works, compile the program, and right click on the bright blue boldface statement names, and the click on "Show Expansion."

RVTEST This program illustrates SLX's 39 built-in random variate generators.

SAFEIFDEF This program shows how to use SLX #safesymbols, which are useful in building programs that use large numbers of #ifdefs and/or #ifndefs. One of the problems when using constructs such as "#ifdef MyName" is that typographical errors such as "#define Myname ON" can produce unintended results where a misspelled symbol is used. SLS allows you to define "safe" symbols, e.g., "#safesymbol MyName", and interrogate them safely, e.g., "#safeifdef MyName". If the symbol queried by a "safeifdef or #safeifndef has not been defines as a #safesymbol, an error message is used. This allows you to define a set of configuration controls and catch any typos that might occur in their interrogation.

SAFESTRINGS In SLX, all string variables have declared maximum lengths. Assigning a value to a string when the value is longer than the declared maximum length results in automatic, "silent" truncation. This program shows how to define several SLX statements that can detect such truncation and issue warnings or error messages when truncation takes place.

INLINEMIN This example shows how to use SLX's "inline" feature in conjunction with a user-defined macro to create a "min(a, b, c, …)" function, which is not included in native SLX.

OBJECTID This example shows how to retrieve the ID and use count for an object. All objects (instances of a given class) are assigned unique serial numbers. All objects also have a use count. A use count reflects the number of ways in which an object can be accessed. When an object's use count goes to zero, it is automatically destroyed. (The SLX "destroy" verb decrements an object's use count and if the use count goes to zwero, destroys it.) Attempting to destroy an object with a use count greater than one results in actual destruction being deferred until the object's use count finally goes to zero.

TYPENAMES This example shows how to determine the types of variables supplied as statement/macro operands.

BEZIER1 This program illustrates the use of Bezier polynomial-based random variate generation. The Bezier control points used were fitted to empirical data, using Mary Ann Flanigan Wagner's Prime software. ExpertFit was used to generate a "named" fit for the same data. The resulting "best fit" distribution was a Weibull distribution. Variates are generated using both techniques. The null hypothesis that the two sets of data came from the same distribution is tested using the Kruskal-Wallace and Mann-Whitney U tests, both of which are built into SLX.

CDFUNC This program illustrates the use of GPSS/H-style discrete and continuous empirical random functions. Ordered pairs defining the CDFs are supplied both in "hard-coded" form and read from a data file.

PAUSE This program shows how a program can pause under Windows (presumably to allow some other program to run).