

# **FROGS**

**Event-driven Simulator**

**Version 1.2**

**SIMEX Programmer's Manual**

**July 2001**

## Overview

SIMEX is an event-driven simulation kernel available as a C++ library which is part of the FROGS environment. FROGS is an object-oriented, general-purpose simulation tool designed for efficient and fast behavioral evaluation of complex architectures. It can be used interactively either as a tool for aiding design and tuning of time-constrained systems or as a decision aid tool for adjusting the behavior of these systems in production.

FROGS exhibits event-driven simulation techniques based on a unique, internally maintained time reference, which is totally independent from the host workstation's idea of time.

FROGS consists of :

- ⑩ SIMEX which provides a rich set of objects required for system modelling, such as threads or activities, synchronization objects, measurement tools and statistical support.
- ⑩ A GUI application for interactive configuration and on-line monitoring of the simulation. The simulation monitor allows you to display and analyze selected parameter evolution, such as state transitions of simulation objects.

# SxDaemon

## PURPOSE

SxDaemon is the superclass of classes implementing local asynchronous actions, which may occur upon the occurrence of state transitions or signals to system objects (i.e. SxObject).

Multiple daemons can be linked together to create an activation list.

This class should be subclassed to implement the `body()` method which is expected to perform the required actions.

## CONSTRUCTOR

SxDaemon(int prio)

Creates a daemon object of priority **prio**. The priority value is used to determine the firing order of multiple daemons linked together in a given activation list. 0 is the lowest priority.

## METHODS

SxDaemon \*addDaemon(SxDaemon \*devil)

Adds daemon **devil** to the linked list of daemons starting from **this**, according to its priority. Highest priority daemons are put at the front of the linked list, lowest are put at the rear. FIFO insertion order applies when two daemons have the same priority. This method returns a pointer to the daemon at the front of the linked list after the insertion is performed.

As a special case, passing NULL to **devil** leads to a null-effect, and always causes the method to return **this**.

virtual void body()

The callback method called when the daemon is fired. Subclasses from the SxDaemon class must reimplement this method to define the proper actions to be performed when activated. The basic callback does nothing.

SxDaemon \*remDaemon(SxDaemon \*devil)

Removes daemon **devil** from the linked list of daemons starting from **this**. This method returns a pointer to the daemon which remains at the front of the linked list after the removal is performed. NULL indicates that no other daemon follows the just removed head of list.

int fire(int ord)

Activates the daemon, calling back its `body()` method. All the daemons eventually linked to **this** using `addDaemon()` are also activated in sequence, according to their priority.

**ord** is the 0-based ordinal position of the current daemon in the activation list. The method returns the actual number of fired daemons.

**SxDaemon \*isLinked(SxDaemon \*devil)**

Checks whether **devil** is currently linked to the activation list headed by **this**. Returns **devil** if true, NULL if not.

**SxDaemon \*next()**

Returns a pointer to the daemon following **this** in the activation list, or NULL if there is none.

**static void setGlobalTrace(int level)**

Set the current trace level of the **SxDaemon** methods to **level**. A value of 1 gives minimal trace information. A value of 2 adds detailed information to the output.

## SxEvent (extends SxObject)

### PURPOSE

SxEvent implements a basic event object, used to signal elementary conditions.

Multiple events can be linked together to create a signaling list. When an event is signaled, each following event from the signaling list it is heading is signaled in turn.

Events have two active states:

- Ⓢ LOW denotes an idle condition.
- Ⓢ HIGH denotes a raised event.

### CONSTRUCTOR

SxEvent(SxDaemon \*handler, SxEvent \*buddy)

Builds a new event object. **handler** is a pointer to a daemon object that will be fired each time the event's internal state switches from LOW to HIGH. **buddy** is a pointer to the event object following **this** one in the signaling list.

### METHODS

void addEvent (SxEvent \*event)

Adds **event** at the end of the signaling list headed by **this**.

SxEvent \*remEvent(SxEvent \*event)

Removes **event** from the signaling list headed by **this**. This method returns a pointer to the event that remains at the front of the linked list after the removal is performed. NULL indicates that no other event follows the just removed head of list.

As a special case, passing NULL to **event** leads to a null-effect, and causes the method to return **this**.

void addHandler(SxDaemon \*handler)

Adds **handler** to the linked list of handlers the event holds. Highest priority handlers are put at the front of the linked list, lowest are put at the rear. FIFO insertion order applies when two handlers have the same priority.

SxDaemon \*remHandler(SxDaemon \*handler)

Removes **handler** from the linked list of handlers the event holds. This method returns **handler** on success, or NULL if the non-null **handler** is not a member of the current activation list.

As a special case, all handlers are removed from the event's handler list in a single operation if handler is a NULL pointer, and the pointer to the handler which was heading the list before the complete removal is returned.

SxDaemon \*isArmed(SxDaemon \*handler)

Checks whether **handler** is currently armed (i.e. member of the activation list) for the event. Returns **handler** if true, NULL if not.

SxEvent \*isLinked(SxEvent \*event)

Checks whether **event** is currently linked to the signaling list headed by **this**. Returns **event** if true, NULL if not.

void link(SxEvent \*event)

Makes **event** follow **this** in the signaling list it is heading.

SxEvent \*next()

Returns a pointer to the event following **this** in the signaling list, or NULL if there is none.

virtual int signal(int state)

Signals the event with **state**. If the event object's internal state switches from LOW to HIGH, the activation list of handlers held by the event is fired. The events eventually linked to **this** are also signaled in sequence with the same input state, according to their insertion order.

static void setGlobalTrace(int level)

Set the current trace level of the SxEvent methods to **level**. A value of 1 gives minimal trace information. A value of 2 adds detailed information to the output.

## SxStateEvent (extends SxEvent)

### PURPOSE

SxStateEvent implements a specialized event object used to signal state changes from SxObject instances. SxObject implementation causes the events associated with system objects to be signaled after each significant state transition.

The state event object must go alternatively through the *off* and *on* states for the event processing handlers to be fired. The expected values used to encode both states are up to the user, and should be passed to the appropriate parameters from the constructor.

### CONSTRUCTOR

SxStateEvent(int onState, int offState, SxDaemon \*handler, SxEvent \*buddy)

Builds a new state event object. **handler** is a pointer to a daemon object that will be fired each time the event's internal state switches from **offState** to **onState**. **buddy** is a pointer to the event object following **this** one in the signaling list.

## SxFlag (extends SxSynchro)

### PURPOSE

**SxFlag** implements a synchronization object allowing a set of threads to wait for a condition. A simulation flag has two possible states, whether ON (i.e. the condition is satisfied) or OFF (i.e. the condition is unsatisfied).

Multiple threads can pend on a single flag until the condition is satisfied.

### CONSTRUCTORS

**SxFlag**(const char \*name, SxSynchroState state =OFF)

Builds a new simulation flag. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

The flag is set to the initial **state** passed to the constructor, whether OFF (default) or ON.

**SxFlag**()

Builds a new anonymous simulation flag, initially set to OFF.

### METHODS

virtual void pend()

Makes the current thread pend for the condition to be satisfied. If the condition is already satisfied when the call is issued, the method returns immediately. Otherwise, the current thread is put in the flag's waiting list and suspended, thus causing a thread switch.

void post()

void setOn()

Sets the flag's internal state to ON, causing any pending thread to resume immediately.

void reset()

void setOff()

Sets the flag's internal state to OFF, causing further pend requests to block the calling thread.

static void setGlobalTrace(int level)

Set the current trace level of the **SxFlag** methods to **level**. A value of 1 gives minimal trace information. A value of 2 adds detailed information to the output.



# SxInfo

## PURPOSE

**SxInfo** is a superclass for information message classes. Messages are usually generated by simulation sources (i.e. **SxSource**) and stored into queues (i.e. **SxQueue**) pending by threads.

Information messages have at least the following attributes:

- ⊗ a priority level, used to order the messages when they are inserted in prioritized queues;
- ⊗ a traffic identifier, which can be used to discriminate message categories when performing statistical measurements on queue throughputs;
- ⊗ a message generation time–stamp.

## CONSTRUCTORS

**SxInfo**(int prio =0, int traffic =0)

Builds a new information message of priority **prio** and a traffic identifier set to **traffic**. The internal time–stamp is set to the value of the global variable **SxClock** when the message is built.

**SxInfo**(const **SxInfo**& src)

Builds a copy of the source information message. The internal time–stamp is set to the value of the global variable **SxClock** when the copy is built.

## METHODS

virtual **SxInfo** \*clone()

Returns a clone of the current information message. However, the internal time–stamp should be reset to the value of the global variable **SxClock** when the message is cloned. This method must be implemented in every subclass extending the **SxInfo** superclass. It is called by the source object (i.e. **SxSource**) to generate copies of the original message template when feeding its destination queue.

**SxInfo** \*copy()

Returns a copy of the current information message. All data members of the original message instance are copied, including the time–stamp, which is left to its original value.

void free()

Removes the current message from the simulation queue it has been posted to, if any.

**SxQueue** \*getQueue() const

Returns a pointer to the simulation queue the current message has been posted to, if any, or NULL if not.

`lTime getTime()`

Returns the current message's generation time-stamp.

`int getTraf() const`

Returns the traffic identifier of the current message.

`void setTraf(int traffic)`

Sets the message's traffic identifier to **traffic**.

## **SxManager (extends SxThread)**

### **PURPOSE**

This class should be instantiated only once during the simulation life–time. It implements a set of services aimed mainly at controlling the simulation initialization and termination phases.

The simulation manager should be created during the early initialization steps of a simulator process. It creates a set of simulation management objects, including a monitor when the interactive simulation mode is in effect. This monitor is running as a separate thread in the simulation system, and establishes a bi–directional communication link between an external GUI application displaying simulation data and the active objects that are monitored in the system.

The simulation manager can parse the initial argument vector given to the `main()` routine to find out additional runtime parameters. A parameter set in the argument vector always overrides the same parameter given to the constructor, when applicable. When the constructor returns, one can retrieve user–defined local arguments – which are left unprocessed by the manager’s option parser – in the updated argument vector.

As a thread, the simulation manager also performs the global sampling of all declared statistical objects concurrently to other threads, or simply sits in an idle wait until the simulation ends if no sampling is required.

### **PUBLIC DATA MEMBERS**

`static SxManager *This`

The pointer to the active manager for the current simulation. This pointer is updated by the `SxManager` constructor.

`static ITime execTime`

The simulation time defined for the simulation. `ZEROTIME` means infinite.

`static ITime warmupTime`

The warm up time defined for the simulation.

`static ITime finishTime`

The sum of the `warmupTime` and `execTime` times, which is the absolute time the executive’s clock should reach for the simulation to end. This value is set to `ZEROTIME` if the simulation is infinite.

`static int numSamples`

The number of statistical samples that will be collected during the execution time, starting after the warm up phase. This value is set to 0 if the simulation is infinite.

`static ITime samplingPeriod`

The duration of a sampling period, which equals `execTime / numSamples` unless the sampling is disabled due to an infinite simulation.

`static ITime samplingTime`

The starting simulation date of the last sampling started. This variable is updated at the beginning of each sampling period.

`static int fMonitored`

A flag telling whether monitoring is active for the simulation. If true, the `SxMonitor::This` class data member should contain a valid pointer to the active monitor.

`static int flnfinite`

A flag telling whether simulation is infinite. If true, `execTime` should be equal to `ZEROTIME`.

`static int fDone`

A flag telling whether simulation has finished. If true, `SxManager::run()` is about to return to its caller, or has already done so.

`static int fRunning`

A flag telling whether simulator has entered its running state. This state is entered by invoking the `SxManager::run()` method.

`static char *progPath`

A null-terminated character string containing the absolute path of the currently executing program. This path is determined inside the `SxManager` constructor, using the value of `argv[0]` as a hint to find it. If the argument vector is not passed, this variable is set to point to an empty string.

## CONSTRUCTOR

`SxManager(ITime simuTime, ITime warmupTime, int *argc = 0, char *argv[] = 0, int nSamples = 1)`

Creates the simulation manager. Once created, the manager should be initialized (`initialize()`) then activated (`run()`) to actually start the simulation.

**simuTime** is an internal time value specifying the total duration of the simulation process. A value of `ZEROTIME` means an infinite simulation time, and disables the sampling procedure.

**warmupTime** is an internal time value giving the duration of the warmup period starting the simulation. The manager will wait for this period to elapse before sampling the declared statistical objects. A value of `ZEROTIME` causes the manager to begin sampling immediately after the simulation is started.

**argc** is a pointer to the argument count passed to the `main()` function. An updated value (lesser or equal to the original) representing the number of remaining options to process

(i.e. left unprocessed by the simulation manager's option parser) is written before the constructor returns. A NULL pointer (default) prevents the option parsing to take place.

**argv** is a pointer to the argument vector passed to the `main()` function. The unprocessed options are trimmed to fill the lower indices of the vector. Hence, iterating from 0 to `argc - 1` after the constructor has returned gives access to the set of unprocessed options. A NULL pointer (default) prevents the option parsing to take place.

**numSamples** is used to determine the sampling period used by the manager. The sampling period is equal to: **simuTime** / **numSamples**. However, if **numSamples** is 0, **simuTime** is forced to **ZEROTIME**, causing the simulation to be infinite. The default value for this parameter is 1.

## COMMAND-LINE ARGUMENTS

The options interpreted by the simulation manager when found in the argument vector passed to the constructor are:

- ⑩ **-x <flags>**, where **flags** is a string of single-letter toggles. “a” should be used to suspend the simulation on trace alerts. This flag is ignored if the simulation is not interactive (i.e. no monitoring support). “m” causes all threads to use a conservative context-switching method, instead of the faster (default) one. This flag is useful when the program code is tweaked by machine-code level instrumenters (such as Purify™) requiring the full context of a suspended thread to be saved. You should try setting this flag each time your simulation crashes unexpectedly at startup after such instrumenter altered the original executable image. “w” causes the simulator to be suspended by the monitor each time a warning message is sent to the simulation manager (i.e. `SxManager::warning()`). For instance, `-Xamw` toggles all options on.
- ⑩ **-p <tcpPort>** where **tcpPort** is a TCP port number a GUI application can pass to the simulator it spawns, in order to connect back to it. The GUI front-end should have previously created a server TCP socket the simulator will connect to, usually on behalf of the invocation of `SxManager::createMonitor()`. The port number can be retrieved using the `SxManager::getTcpPort()` accessor.
- ⑩ **-u <unit>** where **unit** is the default external time unit, “sec” standing for seconds, “msc” for milliseconds and “usc” for microseconds. This unit will be used as the default one when constructing **ETime** objects (see *statobj* package documentation). The **ETime** class implements the external representation of the internal (i.e. **ITime**) simulation time, in which the simulation clock is expressed.
- ⑩ **-k <dtick>** where **dtick** is the display tick value used to scale internal time values when formatting their external representation. **dtick** is a time string composed of a number and a time unit. For instance, passing `-k “1 msc”` causes time values to be displayed in number of milliseconds when formatted.

- ⑩ `-t <simuTime>` where `simuTime` is the total duration of the simulation. When this date is reached, the manager automatically terminates the simulation program, exiting with a zero return code. `simuTime` is a time string composed of a number and a time unit. When set, this option overrides the `simuTime` parameter passed to the constructor.
- ⑩ `-w <warmupTime>` where `warmupTime` is the total duration of the simulation warm up phase. When this date is reached, the manager starts collecting the statistical information according to the sampling period. `warmupTime` is a time string composed of a number and a time unit. When set, this option overrides the `warmupTime` parameter passed to the constructor.
- ⑩ `-s <numSamples>` where `numSamples` is the number of statistical samples to collect during the simulation life-time, after the warm up phase. When set, this option overrides the `numSamples` parameter given to the constructor.
- ⑩ `-d <rundir>` where `rundir` should be a valid path of a directory the simulation process will enter shortly after `SxManager::run()` is invoked. If no run directory is specified, the current directory on entry is kept.
- ⑩ `-l <logFile>` where `logFile` is the path of a text file to which the simulator messages, such as warnings and errors, will be directed to. A value of “-” (i.e. minus) causes the messages to be written to the simulator’s standard error stream, which also happens to be the default setting.
- ⑩ `-z <speedVal>` where `speedVal` is the simulation speed value to set before the simulation starts. The lower the speed value, the greater the hog factor set for to the `SxHog` instance. Hence, lowering the simulation speed makes the hog thread slow down the whole simulation process. The lowest speed value is 1, the highest is 10 (inclusive).
- ⑩ `-C <config>` specifies the name of the configuration (or architecture) which should be instantiated to populate the simulation at startup. Configurations are built using the FROGS monitor.
- ⑩ `-f <project-file>` tells the simulation manager to extract the simulated architecture to instantiate (see `-C` option) from the project file given in argument. A project file is created using the FROGS monitor, and usually ends with the `.frg` extension. In this mode, the monitor becomes a slave of the simulator which spawns it. This mode is useful to run the simulator under the control of a debugger while keeping the monitoring support.

## METHODS

`virtual SxMonitor *createMonitor()`

This callback method should be implemented by subclasses to connect to a GUI application as needed. This application would be expected to display the information obtained from the monitored objects inside the simulation. This double-tasks architecture (simulation backend attached to a graphical front-end, both in separate processes) is recommended, although the monitoring protocol classes (e.g. `GraphProto`) do not specifically enforce it.

`createMonitor()` should return a pointer to an instance of a `SxMonitor` subclass, which will be remembered as the global monitor object for the simulation. Returning `NULL` causes the executive to switch to the non-interactive mode.

The default implementation of this method attempts to connect to an external program using the current value of the TCP port number (see the discussion about the command-line argument `-p`). If it succeeds, a basic `SxMonitor` class instance is created and a pointer to it is returned to the caller.

**virtual void initialize()**

Initializes the manager before the simulation is started. This method does the following tasks:

- ④ the error log stream is opened. One should note that the run directory which may have been specified through the command line options (i.e. `-d` flag) has not been entered yet, when evaluating relative file pathes;
- ④ if the simulation is interactive, the monitor is created through a call to `SxManager::createMonitor()` and the communication channel is opened with its display application;
- ④ the simulation is populated by instantiating the current architecture (i.e. `-C` flag);
- ④ the hog thread is created (see `SxHog`).

This method should be called on behalf of the `main()` thread, usually shortly after the manager object is created, and before `SxManager::run()` is invoked to start the simulation.

**virtual int run()**

This method actually starts the simulation process, yielding the processor control to the thread having the earliest scheduling time. An integer status code is returned to the caller after the simulation ends, whether normally (zero exit code) or abnormally (non-zero exit code).

As a consequence of starting a monitored simulation, the `StProto::protoInit()` method is called for all the instances of classes extending the `StProto` class which were created during the configuration phase (i.e. `SxManager::initialize()`).

This method *must be called* on behalf of the `main()` thread context.

**virtual void finish(int exitCode)**

Terminates the simulation. The control is switched to the `main()` thread, causing `SxManager::run()` to return with the termination status **exitCode**. `finish()` should be the preferred way of terminating a simulation, rather than calling the C-level `exit()` routine directly.

**int getTcpPort() const**

Returns the TCP port number passed to the simulator through the command-line arguments (i.e. `-p` option), or `-1` if none has been given.

`int getFatalCount() const`

Returns 1 or 0, whether a fatal error occurred or not.

`int getWarningCount() const`

Returns the number of emitted warning messages, which corresponds to the number of times the `warning()` method has been called.

`virtual void fatal(const char *format, ...)`

Formats then emits an abort message to the error log file. **format** is a format string conforming to the `printf(3S)` specifications. An adequate variable argument list should follow this parameter. The control is switched to the `main()` thread after the message is logged, causing `SxManager::run()` to return with an abnormal termination status of 1.

`virtual void warning(const char *format, ...)`

Formats then emits a warning message to the error log file. **format** is a format string conforming to the `printf(3S)` specifications. An adequate variable argument list should follow this parameter. The count of warnings is incremented.

The simulator may be suspended by the monitor (if active) after the message is reported if the “w” (i.e. break on warnings) option flag was passed to the simulation manager (see `SxManager::SxManager()`).



## SxMonitor (extends SxThread)

### PURPOSE

This class should be instantiated only once during the simulation life–time. It implements a set of services aimed at providing monitoring capabilities. Monitoring should be understood as an ability to interact with the internal simulation objects from an external application during the simulation life–time, such as displaying or eventually changing the objects’ internal state. For instance, threads behavior against time can be displayed using a state diagram while the simulation monitor exports each state transition automatically.

The embedded monitor runs as a separate thread in the simulation; it is mainly in charge of waiting for commands from the external (usually the GUI) application, listening to a TCP socket for input. The monitor uses the **StProto** communication scheme to dispatch received messages and send information to the display front–end. Describing the specific protocol values (such as message identifiers and structs) currently shared between SIMEX and the ISE is outside the scope of this document. The information given below only describes the general way of using the simulation monitor as a communication end–point between the internal objects managed by the executive and a display front–end.

The monitor is only created while running in interactive mode; when enabled, it is also in charge of holding and releasing the simulation process as requested by the display front–end. The monitor uses the callout management service from the **SxScheduler** class to register itself for asynchronous input notification from the communication channel.

The simulation monitor should be created in response to the invocation of **SxManager::createMonitor()** by the simulation manager.

### CONSTRUCTOR

**SxMonitor(SxTcp \*tcpChannel)**

Creates a monitor object using **tcpChannel** as a pointer to a TCP communication object for sending and receiving data to/from the display front–end.

### PUBLIC DATA MEMBERS

static **SxMonitor \*This**

The pointer to the active monitor for the current simulation. This pointer is updated by the **SxMonitor** constructor. This pointer may be null if the simulation is not started in interactive mode.

### METHODS

virtual void **send(int mtype, StProtoMessage \*mbuf, int msize)**

Sends the message of type **mtype**, starting from **mbuf** to **msize** bytes to the TCP channel associated with the monitor object. See the documentation of the **StProto** communication scheme for more on these parameters.

`virtual void stopSimulation(int stopCondition)`

Stops the simulation, sending the **stopCondition** word to the display front-end as an indication of the cause of the suspension. This method immediately switches control to the monitor thread, which in turn calls `SxMonitor::holdSimulation()` as a consequence of entering the stopped state.

`virtual void holdSimulation()`

Issues a blocking read request on the communication channel to hold the simulation process. Because the SIMEX multi-threading is based on co-routines, the overall process is blocked until a release message is sent from the ISE to the embedded monitor.

`virtual void terminate()`

Emits a finalization message to the display front-end before entering some kind of “zombie” state, allowing the front-end to request information to the embedded monitor through the communication channel, usually for post-mortem analysis purposes. At this point, all other threads are suspended. This callback method is called from the simulation manager, after the simulation has finished but before `SxManager::run()` returns to its caller.

## SxObject (extends StStateDiagram)

### PURPOSE

**SxObject** is the root class for most of the SIMEX simulation object classes. This class implements the basic simulation system object. This object can have multiple programmable states the simulation monitor can export to a display front-end upon transition from one state to another. For instance, timed objects are system objects managed by the executive's scheduler which can be idle, pending on a resource, preempted by another object or running. State transition logging is performed by the inherited **StStateDiagram** class implementation.

**SxObject** instances can hold a list of **SxStateEvent** instances which will be signaled each time a significant state transition occurs at the system object level.

### CONSTRUCTOR

**SxObject**(const char \*name, int pflags =0, int logSize =NSTVALUEDEF)

Creates a system object. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

**pflags** is a set of protocol flags such as defined by the **StProto** interface.

**logSize** is the size of the state log for the object. This value is used to determine the number of consecutive state transitions the simulation executive will keep in memory for this object at any time, specifically for statistical and monitoring purposes. NSTVALUEDEF is currently 100.

### METHODS

int getState() const

Returns the current object's state.

int setState(int state)

Sets the object's new internal state to **state**. If the new state differs from the current one, the associated state event objects are immediately signaled with the new state. Finally, the new state value is logged at the state diagram level, which in turn eventually sends it to the display front-end. The total number of fired signal handlers is returned.

**SxStateEvent** \*armEvent(int onState, int offState, **SxDaemon** \*handler)

Creates and associates a new state event object with the current system object. The **SxStateEvent** object is instantiated in order to call **handler** each time the system object's internal state switches from **offState** to **onState**. A pointer to the new state event object is returned.

**SxStateEvent** \*armEvent(**SxStateEvent** \*event, **SxDaemon** \*handler)

Ensures that **event** is currently armed for the current system object, inserting it in the object's event list if necessary. The daemon object **handler** is then armed for **event**. If more than one event objects are headed by **event**, all are inserted in the system object's event list, but only **event** gets **handler** armed for it.

SxDaemon \*remEvent(SxStateEvent \*event, SxDaemon \*handler)

If **event** is armed for the current system object and **handler** is armed for **event**, this method disarms **handler** and returns it.

If **event** is armed for the current system object and **handler** is NULL, this method removes **event** from the system object's event list and returns the first handler which was armed for it.

As a special case, passing a NULL **event** leads to a null-effect and always returns NULL.

SxStateEvent \*isArmed(int onState, int offState, SxDaemon \*handler =0) const

Searches for a state event associated with the current system object monitoring transitions from **onState** to **offState**. If **handler** is non-zero, it must be armed for the found event. A pointer to a matching event is returned on success, otherwise NULL is returned if no state event was found.

The first event matching the conditions found in the system object's event list is retained, even if more than one could satisfy them.

SxStateEvent \*isArmed(SxStateEvent \*event, SxDaemon \*handler =0) const

Checks whether **event** is a member of the current system object's event list. If **handler** is non-zero, it must be armed for **event**. **event** is returned if successful, otherwise NULL if no matching **event** was found.

SxStateEvent \*isArmed(SxDaemon \*handler) const

Returns a pointer to the first state event from the current system object's event list for which **handler** is armed. NULL is returned if no matching event is found.

static void setGlobalTrace(int level)

Set the current trace level of the **SxObject** methods to **level**. A value of 1 gives minimal trace information. A value of 2 adds detailed information to the output.

virtual int stateIndex(int state)

This method should be implemented by subclasses if the system object's state values need to be rescaled before sending them to the state diagram superclass. **SxObject::signal()** typically calls this method with the new entered state before passing it to the **StStateDiagram::add()** method for logging and export (i.e. to the display front-end).

For instance, one could need to map states **DEAD** =3 and **KILLED** =7 to the display state "DORMANT" indexed on value 0. So this method could be reimplemented as in the following one:

```

int SomeTaskObject::stateIndex (int _state)
{
    if (_state == DEAD || _state ==KILLED)
        return 0;
    return _state;
}
void SomeTaskObject::protoInit ()
{
    const char *stateArray[3];
    stateArray[0] = "DORMANT";
    stateArray[1] = "...";
    ...
}

```

## SxQueue (extends SxSynchro)

### PURPOSE

This class implements a thread synchronization object allowing threads to wait for messages. Messages are prioritized, and must be subclasses of the **SxInfo** superclass.

A queue has four active states:

- ⑩ **PENDED** indicates that at least one thread is pending for messages on input. This state denotes an empty queue.
- ⑩ **OFF** means that the queue is idle, with neither messages stored, nor pending consumer threads.
- ⑩ **ON** means that at least one message is available to consumer threads, but no output contention exists.
- ⑩ **POSTED** means that an output contention currently exists on the queue, which needs to dispatch available messages before accepting further posting from suspended producer thread(s).

### CONSTRUCTORS

**SxQueue**(const char \*name, InsertMode qmode, unsigned msgMax =SXQUEUE\_MAXMSG, InsertMode pmode =FIFO)

Creates a message queuing object. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

Messages are queued following the **qmode** insertion order, which may be one of the following:

- ⑩ **FIFO** requests a first-in, first-out insertion.
- ⑩ **LIFO** requests a last-in, first-out insertion.
- ⑩ **PRUP** (or **PRUPFF**) applies an insertion rule driven by increasing priority of messages (lowest priority value first). When two messages have the same priority, FIFO ordering is applied.
- ⑩ **PRUPLF** applies an insertion rule driven by increasing priority of messages. When two messages have the same priority, LIFO ordering is applied.
- ⑩ **PRDN** (or **PRDNFF**) applies an insertion rule driven by decreasing priority of messages (highest priority value first). When two messages have the same priority, FIFO ordering is applied.
- ⑩ **PRDNLF** applies an insertion rule driven by increasing priority of messages. When two messages have the same priority, LIFO ordering is applied.

A maximum of **maxMsg** can be queued before producer threads are suspended until queued messages are consumed, and the number of messages waiting to be read falls

under this limit. The special value `SXQUEUE_MAXMSG` can be used to indicate a pseudo-infinite ( $2^{32} - 1$ ) limit.

Threads pend on the queue object according to the **pmode** order, whether they are producer threads suspended due to an output contention, or consumer threads waiting for input messages. This mode may be one of the following:

- ⑩ FIFO ensures that the oldest pending thread is always served first.
- ⑩ LIFO ensures that the latest pending thread is always served first.
- ⑩ PRUP (or PRUPFF) ensures that the thread having the lowest priority is always served first. When two threads have the same priority, the oldest pending one is priority.
- ⑩ PRUPLF ensures that the thread having the lowest priority is always served first. When two threads have the same priority, the latest pending one is priority.
- ⑩ PRDN (or PRDNFF) ensures that the thread having the highest priority is always served first. When two threads have the same priority, the oldest pending one is priority.
- ⑩ PRDNLF ensures that the thread having the highest priority is always served first. When two threads have the same priority, the latest pending one is priority.

`SxQueue(InsertMode qmode, unsigned msgMax =SXQUEUE_MAXMSG, InsertMode pmode =FIFO)`

Creates an anonymous message queuing objects. The parameters are the same as described for the previous constructor.

`SxQueue()`

Creates an anonymous message queuing object, with default parameters, such as calling this constructor is identical to invoke:

```
new SxQueue(FIFO,SXQUEUE_MAXMSG,FIFO);
```

## METHODS

`void post(SxInfo *msg)`

Posts a message to the current queue object. **msg** is a pointer to an instance of a subclass of the `SxInfo` superclass. If a thread is currently pending for a message on this queue, it is dispatched the message and resumed immediately, thus preempting the caller.

If no thread is waiting for messages, **msg** is inserted according to the queue's insertion mode. This insertion is done immediately, unless the queuing limit has been reached, in which case the calling thread is suspended until a message slot is freed by a consumer thread. This special case denotes an output contention. A producer thread resumed after an output contention does not preempt the consumer thread clearing the condition.

`void postFront(SxInfo *msg)`

This method is a variant of `post()` which bypasses the queue's message insertion mode. The jammed **msg** is posted at front of the message list.

### **SxInfo \*get()**

Returns the first available message from the queue. The calling thread is suspended until the message comes in if the queue is empty. The incoming message is unlinked from the queue before its address is returned to the caller.

If pend hooks exist for the current queue, they are fired immediately before the first available message is extracted from the queue.

### **SxInfo \*accept ()**

Returns the first available message from the queue, or NULL if no message is immediately available. The incoming message is unlinked from the queue before its address is returned to the caller.

If pend hooks exist for the current queue, they are fired immediately before the first available message is extracted from the queue.

### **void remove(SxInfo \*msg)**

Removes a message from the queue. If **msg** is valid and currently linked to the queue, pend hooks (if any) are fired immediately before the message is removed. Passing NULL or the address of a message not linked to the queue leads to a null-effect.

### **SxInfo \*first()**

Returns the address of the first message available from the current queue to consumer threads, or NULL if none.

### **SxInfo \*last()**

Returns the address of the last message available from the current queue to consumer threads, or NULL if none.

### **unsigned getOCount() const**

Returns the count of messages available from the current queue to consumer threads.

### **unsigned getOMax() const**

Returns the current queue's output contention threshold (in number of messages).

### **StObject \*setStatistics(SxStatisticType type =STAT\_MEAN)**

Creates and returns the address of a statistical object which will integrate the number of messages in the current queue over the time. In the current implementation, **type** must be STAT\_MEAN.

### **static void setGlobalTrace(int level)**

Set the current trace level of the **SxQueue** methods to **level**. A value of 1 gives minimal trace information. A value of 2 adds detailed information to the output.



## SxResource (extends SxSynchro)

### PURPOSE

This class implements a thread synchronization object acting like a semaphore. Thread execution can be serialized using this object, by allowing a limited number of resources to be dispatched to them. The lack of resource causes the requesting thread to pend on the object until one becomes available.

A resource has three active states:

- ⑩ **PENDED** indicates that at least one thread is pending for a resource. This state denotes that no resource unit is currently available.
- ⑩ **OFF** means that the resource is idle, with neither units, nor pending threads.
- ⑩ **ON** means that at least one resource unit is available to threads.

### CONSTRUCTOR

**SxResource**(const char \*name =0, unsigned initCount =0, InsertMode pmode =FIFO, InsertMode hmode =FIFO, int fPreempt = 0)

Creates a resource object. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

**initCount** specifies the number of units initially available for the resource.

Threads pending for a resource availability are queued according to **pmode**, which may take one of the following values:

- ⑩ **FIFO** ensures that the oldest pending thread is always served first.
- ⑩ **LIFO** ensures that the latest pending thread is always served first.
- ⑩ **PRUP** (or **PRUPFF**) ensures that the thread having the lowest priority is always served first. When two threads have the same priority, the oldest pending one is priority.
- ⑩ **PRUPLF** ensures that the thread having the lowest priority is always served first. When two threads have the same priority, the latest pending one is priority.
- ⑩ **PRDN** (or **PRDNFF**) ensures that the thread having the highest priority is always served first. When two threads have the same priority, the oldest pending one is priority.
- ⑩ **PRDNLF** ensures that the thread having the highest priority is always served first. When two threads have the same priority, the latest pending one is priority.

If **fPreempt** is non-zero, threads can preempt themselves for the resource, and **hmode** gives the preemption order. This parameter may take one of the following values:

- ⑩ **FIFO** ensures that the oldest pending thread is preempted before latest ones.
- ⑩ **LIFO** ensures that the latest pending thread is preempted before oldest ones.

- ⑩ PRUP (or PRUPFF) ensures that the thread having the lowest priority is preempted by others. When two threads have the same priority, the oldest pending one is preempted.
- ⑩ PRUPLF ensures that the thread having the lowest priority is preempted by others. When two threads have the same priority, the latest pending one is preempted.
- ⑩ PRDN (or PRDNFF) ensures that the thread having the highest priority is preempted by others. When two threads have the same priority, the oldest pending one is preempted.
- ⑩ PRDNLF ensures that the thread having the highest priority is preempted by others. When two threads have the same priority, the latest pending one is preempted.

## METHODS

`void request()`

`void pend()`

Attempts to dispatch a resource unit to the calling thread. If the currently available unit count is non-zero, it is decremented and the method returns immediately. Otherwise, the following actions may take place:

- ⑩ the current resource object is marked as preemptable, and the most preemptable thread among the current one and those currently owning units from this object is selected, according to the preemption order. If the selected thread differs from the calling one, it is immediately preempted and the caller returns. It should be noted that no provision is made to resume the execution of the preempted thread.
- ⑩ the current source object is not marked as preemptable, or the most preemptable thread is the current one, then the calling thread is suspended until a resource unit becomes available.

If pend hooks exist on the current object, they are fired immediately before the method returns.

`void release()`

`void post()`

Releases a resource unit. If a thread is currently waiting for a unit to become available, it is resumed and the caller returns immediately. Otherwise, the count of resource units is incremented.

If post hooks exist on the current object, they are fired immediately before the method returns.

`unsigned getOCount() const`

Returns the count of resource units available from the current object.

`void setPreemptable(InsertMode hmode, int fPreempt)`

Changes the preemptability state of the resource object. **hmode** and **fPreempt** both specify the new setting, and have the same meaning than the constructor parameters. This method does not reorder the current preemption list contents, but rather sets the new mode for subsequent updates.

**StObject \*setStatistics(SxStatisticType type =STAT\_MEAN)**

Creates and returns the address of a statistical object which will integrate the number of units in the current resource over time. In the current implementation, **type** must be **STAT\_MEAN**.

**static void setGlobalTrace(int level)**

Set the current trace level of the **SxResource** methods to **level**. A value of 1 gives minimal trace information. A value of 2 adds detailed information to the output.

## SxSource (extends SxTimed)

### PURPOSE

This class is the superclass of message-generating objects. The behavior of a source consists in producing messages at specific times determined by a private generation law. Message types must be subclasses of the **SxInfo** superclass. Once generated, a message is automatically posted to a destination queue (i.e. **SxQueue**). The insertion order is controlled by the message priority.

A pre-defined set of typical subclassed sources is available with SIMEX, including periodical (**SxPerSource**), exponential (**SxExpSource**), uniform (**SxUniSource**) and file-based (**SxFileSource**) sources.

### CONSTRUCTOR

**SxSource**(**SxInfo** \*msgTempl, **StNumericLaw** \*law, **SxQueue** \*destq, int ngen =1, **ITime&** tStart =ZEROTIME, **ITime&** tEnd =MAXITIME, int prio =0, int fAutoDel =0)

Creates a message source object. **msgTempl** will be used by the source as a template to produce the messages posted to the destination queue **destq**. The virtual method **SxInfo::clone()** will be invoked to obtain a cloned instance of the template each time it is necessary. The *message template is deleted by the source object* as a part of its destructor actions, so the required way of initializing a source is to create a new template object each time.

The source is only active between the given time bounds specified by **tStart** and **tEnd**. After **tEnd** is reached, the source calls its **finalize()** virtual method to perform its cancellation. The default action of this method is to delete the calling object if the **fAutoDel** flag was passed when building the source. One may reimplement this method in subclasses to have different housekeeping policies for sources.

After a message is posted to the destination queue, the source determines the next arrival date by calling the **get()** virtual method of the **law** object, which return value is added to the current contents of the global **SxClock** variable (i.e. the numeric law is expected to return positive or null time increments relative to the simulation clock). **ngen** messages are cloned and posted to the destination queue on arrival. Insertion order of messages in the destination queue may be affected by the **prio** parameter defining the auto-generated messages priority.

If **tStart** is a negative time value, the source will remain indefinitely idle. If **tStart** is ZEROTIME, a starting date is drawn from the **law** object to determine the first message arrival date. Otherwise, the first message will be generated exactly at **tStart**.

### DESTRUCTOR

virtual ~**SxSource**()

Deletes the message template passed to the constructor, and unlinks the source from the manager's source chain.

## METHODS

virtual void finalize()

This callback method is invoked by the source object itself to perform its cancellation when its time limit is reached. The default action of this method is to delete the calling object if the **fAutoDel** flag was passed when building the source. It should be reimplemented for subclasses having different housekeeping policies.

static void setGlobalTrace(int level)

Set the current trace level of the **SxSource** methods to **level**. A value of 1 gives minimal trace information. A value of 2 adds detailed information to the output.

## SxPerSource (extends SxSource)

### PURPOSE

This class defines a periodical message source. Messages are generated at times determined by a periodical numeric law.

### CONSTRUCTOR

SxPerSource(ITime period, SxInfo \*msgTempl, SxQueue \*destq, int ngen =1, ITime& tStart =ZEROTIME, ITime& tEnd =MAXITIME, int prio =0)

Creates a periodical message source object. **msgTempl** will be used by the source as a template to produce the messages posted to the destination queue **destq**. The virtual method **SxInfo::clone()** will be invoked to obtain a cloned instance of the template each time necessary. The *message template is deleted by the source object* as part of its destructor actions, so the required way of initializing a source is to create a new template object each time a new source is built.

The source is only active between the given time limits specified by **tStart** and **tEnd**. After **tEnd** is reached, the source is automatically deleted.

Message arrival dates are based on the **period** value. **ngen** messages are cloned and posted **to** the destination queue on arrival. Insertion order of messages in the destination queue may be affected by the **prio** parameter defining the auto-generated messages priority.

If **tStart** is a negative time value, the source will remain indefinitely idle. If **tStart** is ZEROTIME, the starting date will be **SxClock + period**. Otherwise, the first message will be generated exactly at **tStart**.

## SxExpSource (extends SxSource)

### PURPOSE

This class defines an exponential message source. Messages are generated at times determined by an exponential numeric law.

### CONSTRUCTOR

SxExpSource(ITime mean, SxInfo \*msgTempl, SxQueue \*destq, int ngen =1, ITime& tStart =ZEROTIME, ITime& tEnd =MAXITIME, int prio =0)

Creates an exponential message source object. **msgTempl** will be used by the source as a template to produce the messages posted to the destination queue **destq**. The virtual method **SxInfo::clone()** will be invoked to obtain a cloned instance of the template each time necessary. The *message template is deleted by the source object* as part of its destructor actions, so the required way of initializing a source is to create a new template object each time a new source is built.

The source is only active between the given time limits specified by **tStart** and **tEnd**. After **tEnd** is reached, the source is automatically deleted.

Message arrival dates are based on the outcomes of an exponential numeric law of mean **mean**. **ngen** messages are cloned and posted **to** the destination queue on arrival. Insertion order of messages in the destination queue may be affected by the **prio** parameter defining the auto-generated messages priority.

If **tStart** is a negative time value, the source will remain indefinitely idle. If **tStart** is ZEROTIME, the starting date will be obtained by adding the next outcome of the exponential law to the content of **SxClock**. Otherwise, the first message will be generated exactly at **tStart**.

## SxUniSource (extends SxSource)

### PURPOSE

This class defines a uniform message source. Messages are generated at times determined by a uniform numeric law.

### CONSTRUCTOR

SxUniSource(ITime min, ITime max, SxInfo \*msgTempl, SxQueue \*destq, int ngen =1, ITime& tStart =ZEROTIME, ITime& tEnd =MAXITIME, int prio =0)

Creates a uniform message source object. **msgTempl** will be used by the source as a template to produce the messages posted to the destination queue **destq**. The virtual method **SxInfo::clone()** will be invoked to obtain a cloned instance of the template each time necessary. The *message template is deleted by the source object* as part of its destructor actions, so the required way of initializing a source is to create a new template object each time a new source is built.

The source is only active between the given time limits specified by **tStart** and **tEnd**. After **tEnd** is reached, the source is automatically deleted.

Message arrival dates are based on the outcomes of a numeric law producing values uniformly distributed in the range [**min** .. **max**]. **ngen** messages are cloned and posted **to** the destination queue on arrival. Insertion order of messages in the destination queue may be affected by the **prio** parameter defining the auto-generated messages priority.

If **tStart** is a negative time value, the source will remain indefinitely idle. If **tStart** is ZEROTIME, the starting date will be obtained by adding the next outcome of the uniform law to the contents of **SxClock**. Otherwise, the first message will be generated exactly at **tStart**.



## SxFileSource (extends SxSource)

### PURPOSE

This class defines a file-based message source. Message arrival dates are stored in a text file in increasing order.

### CONSTRUCTORS

SxFileSource(const char \*fileName, SxInfo \*msgTempl, SxQueue \*destq, int ngen =1, int prio =0)

Creates a file-based message source object. **msgTempl** will be used by the source as a template to produce the messages posted to the destination queue **destq**. The virtual method **SxInfo::clone()** will be invoked to obtain a cloned instance of the template each time necessary. The *message template is deleted by the source object* as a part of its destructor actions, so the required way of initializing a source is to create a new template object each time a new source is built.

Message arrival dates are read from the file named **fileName**. **ngen** messages are cloned and posted **to** the destination queue on arrival. Insertion order of messages in the destination queue may be affected by the **prio** parameter defining the auto-generated messages priority.

If **tStart** is a negative time value, the source will remain indefinitely idle. If **tStart** is **ZEROTIME**, the starting date will be obtained by adding the next outcome of the file law to the contents of **SxClock**. Otherwise, the first message will be generated exactly at **tStart**.

Please refer to the documentation of the **StFileLaw** class for a description of the expected file format.

## SxSampler (extends SxThread)

### PURPOSE

This class implements a simple sampler thread synchronizing the activity of a filter object. The sampler triggers a filter update periodically during the simulation life-time. Samplers are privately used by time graphs (i.e. **SxTimeGraphs**) for collecting data periodically from the statistical object it monitors.

### CONSTRUCTOR

**SxSampler**(const char \*name, SxFilter \*filter)

Creates a sampler. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

The sampling thread fires the **SxFilter::update()** virtual method of the **filter** object on a periodical basis. The period is initially obtained from the filter through a call to the **SxFilter::getDtUpdate()** method.

The first update occurs immediately after the sampler starts running.

## SxTimeGraph (extends StTimeGraph)

### PURPOSE

This class implements a measurement object designed to grasp the temporal evolution of a given statistical object all along the simulation. The collected results can be exported to the display front-end on-the-fly, whenever the simulation monitor is active.

The actual computation is performed by the general **StTimeGraph** superclass; however, the **SxTimeGraph** subclass integrates these computations in the context of an event-driven simulation made of timed objects, and connects the time graph to the communication channel held by the simulation monitor.

### CONSTRUCTORS

**SxTimeGraph**(const char \*name, StObject \*so, lTime dtUpdate, StValueType type =VAL, int logSize =NTGVALUEDEF)

Creates a time graph. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

The time graph will collect the value corresponding to **type** from the statistical object **so**, on a periodical basis fixed by **dtUpdate**. **type** must be one of the valid enumerated values one may pass to **StObject::getValue()**. Please refer to the documentation of the **StObject** class for more information on this parameter.

**logSize** is the size of the sample log for the object. This value is used to determine the number of consecutive samples the simulation executive will keep in memory for this object at any time, specifically for statistical and monitoring purposes. NTGVALUEDEF is currently 500.

A sampler thread (i.e. **SxSampler**) is started for the current object.

### DESTRUCTOR

virtual ~**SxTimeGraph**()

Cancels the sampler thread associated with the current object.

## SxHistogram (extends StHistogram)

### PURPOSE

This class implements a measurement object which determines the probability density of any statistical law, with computation of mean and standard deviation and accuracy evaluation for a given confidence interval. The collected results can be exported to the display front-end on demand, whenever the simulation monitor is active.

The actual computation is performed by the general **StHistogram** superclass; however, the **SxHistogram** subclass integrates these computations in the context of an event-driven simulation made of timed objects, and connects the histogram to the communication channel held by the simulation monitor.

### CONSTRUCTORS

**SxHistogram**(const char \*name, int nbins, double leftb, double rightb, StHistAdjustMode mode =MULTIPLY)

Creates an histogram with floating-point bounds, namely **leftb** and **rightb**. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end. The number of counting bins is specified by **nbins**.

Histograms can automatically adjust themselves to the actual range of entered values. The expected behavior when a value falls outside the current range of an histogram is selectable by an adjust **mode** setting:

- Ⓢ **MULTIPLY** causes the histogram range to be adjusted by successive multiplications by 2, either to the left or to the right according to the bound which is overshooted. This is the default mode.
- Ⓢ **GARBAGE** prevents the histogram range to be adjusted, all the values falling outside this range are collected in the leftmost and rightmost bins.

**SxHistogram**(const char \*name, int nbins, int leftb, int rightb, StHistAdjustMode mode =MULTIPLY)

Creates an histogram with integer bounds, namely **leftb** and **rightb**. Other parameters have the same meaning than previously.

The right bound and the number of counting bins are adjusted to obtain an integer bin size.

## SxScaler (extends StObjectScaler)

### PURPOSE

This class implements a measurement object designed to compute and report the division of a given value type of a *scaled* statistical object by a given value type of a *scaling* statistical object. The collected results can be exported to the display front-end on-the-fly, whenever the simulation monitor is active.

The actual computation is performed by the general **StObjectScaler** superclass; however, the **StScaler** subclass integrates these computations in the context of an event-driven simulation made of timed objects, and connects the scaler to the communication channel held by the simulation monitor.

### CONSTRUCTORS

**SxScaler**(const char \*name, StObject \*scaledObject, StObject \*scalingObject, StValueType vtScaled =VAL, StValueType vtScaling =VAL, int pflags =0)

Creates a statistical object scaler. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

**scaledObject** is a pointer to the scaled statistical object, and **scalingObject** is a pointer to the scaling one.

**vtScaled** and **vtScaling** specify the type of values used in the scaling process. These types must be compatible with the computation available from the scaled and scaling objects respectively. Typical settings are:

- ⊗ VAL stands for last entered value.
- ⊗ NUM is the number of aggregated values.
- ⊗ SUM represents the global sum of all collected values since the beginning of measure.

**pflags** is a set of protocol flags such as defined by the **StProto** interface.

## **SxCounter (extends StCounter)**

### **PURPOSE**

This class implements a measurement object designed to sum numerical values, while sampling their sum all along the simulation. The collected results can be exported to the display front-end on-the-fly, whenever the simulation monitor is active.

The actual computation is performed by the general **StCounter** superclass; however, the **SxCounter** subclass integrates these computations in the context of an event-driven simulation made of timed objects, and connects the counter to the communication channel held by the simulation monitor.

### **CONSTRUCTORS**

**SxCounter**(const char \*name, int pflags =0)

Creates a counter. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

**pflags** is a set of protocol flags as defined in the **StProto** interface.

## SxCounterGroup (extends StCounterGroup)

### PURPOSE

This class implements a group of statistical counters gathered under a common name. The collected results can be exported to the display front-end on-the-fly, whenever the simulation monitor is active.

Please refer to the documentation of the **StObjectGroup** class for a detailed information about the available methods.

The actual computation is performed by the general **StCounterGroup** superclass; however, the **SxCounterGroup** subclass integrates these computations in the context of an event-driven simulation made of timed objects, and connects the counter group to the communication channel held by the simulation monitor.

### CONSTRUCTORS

**SxCounterGroup**(const char \*name, const char \*indexName, int nitems, int iBase =0, int pflags =0)

Creates a group of counters. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

**indexName** is a null-terminated character string representing the name of the varying parameter of the group.

**nitems** specifies the number of counters in the group. The constructor builds as many **StCounter** instances as required by **nitems**, plus an extra one which manages the sum of all the others. The global sum counter is given the internal index #0.

**iBase** is the the lowest index value for the group. Indices passed to methods requiring them will be scaled to take in account this threshold. For instance, passing an index base of -2 for a 4-elements group allows using indices -2, -1, 0 (global value), 1 and 2.

**pflags** is a set of protocol flags such as defined by the **StProto** interface.

## **SxTimeIntegrator (extends StIntegrator)**

### **PURPOSE**

This class implements a measurement object designed to automatically compute the integration of a given variable against the simulated time. The collected results can be exported to the display front-end on-the-fly, whenever the simulation monitor is active.

The actual computation is performed by the general **StIntegrator** superclass; however, the **SxTimeIntegrator** subclass integrates these computations in the context of an event-driven simulation made of timed objects, and connects the integrator to the communication channel held by the simulation monitor.

### **CONSTRUCTORS**

**SxIntegrator**(const char \*name, int pflags =0)

Creates a time integrator. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

**pflags** is a set of protocol flags as defined in the **StProto** interface.



## SxSynchro (extends SxObject)

### PURPOSE

This pure class is the superclass of thread synchronization objects. Synchronization objects are tightly coupled with simulation threads (i.e. **SxThread**), as this class implements the basic functionalities for them to wait for and set conditions.

Threads waiting for a condition are put to sleep, then resumed when the synchronization object is signaled, thus indicating that the condition is met.

A synchronization object has four basic states:

- ④ **PENDED** indicates that a thread (at least) is waiting for the condition to be met.
- ④ **OFF** means that the object is idle, the condition being unsatisfied but with no threads waiting for it.
- ④ **ON** means that the condition is met. This implies that no threads are currently waiting for it.
- ④ **POSTED** is a variant of the **ON** state the subclasses can use to exhibit an additional status to the mere condition availability. For instance, the message queue object (i.e. **SxQueue**) enters the **POSTED** state when an output contention occurs, which means that messages are available for reading by consumer threads, but producer threads cannot send more messages until some are effectively consumed.

### PROTECTED DATA MEMBERS

**SxThreadGList** *pendList*

The list of currently pending threads, waiting for the condition to be met.

**SxDaemon** *\*pendHook*

A pointer to the **SxDaemon** object heading the list of triggers which should be fired in turn each time a pend request is issued to the object.

**SxDaemon** *\*postHook*

A pointer to the **SxDaemon** object heading the list of triggers which should be fired in turn each time the condition is signaled to the object.

### CONSTRUCTOR

**SxSynchro**(const char \*name =0)

Creates a synchronization object. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; **NULL** is an acceptable value, preventing the anonymous object from being visible from the display front-end.

### DESTRUCTOR

virtual ~SxSynchro()

The destructor checks for pending threads before returning. If some threads are still waiting for the destroyed object to be signaled, the `alert()` virtual method is fired, and a warning message is sent to the simulation manager.

## METHODS

int getPCount() const

Returns the number of threads currently pending on the object.

void addPendHook(SxDaemon \*devil)

Adds the daemon object **devil** to the list of triggers which should be fired each time a pend request is issued to the current object. The exact context in which those triggers are fired depends on the subclassed implementation for this functionality. The `SxSynchro` object only manages the registration of such triggers.

Multiple calls to this method cause all the daemons to be linked in a single activation list, ensuring that all of them will get called in turn for each event, in a first-in first-out order.

void remPendHook(SxDaemon \*devil)

Removes the daemon object **devil**, previously added by a call to `addPendHook()`, from the list of triggers.

void addPostHook(SxDaemon \*devil)

Adds the daemon object **devil** to the list of triggers which should be fired each time the condition is signaled to the current object. The exact context in which those triggers are fired depends on the subclassed implementation for this functionality. The `SxSynchro` object only manages the registration of such triggers.

Multiple calls to this method cause all the daemons to be linked in a single activation list, ensuring that all of them will get called in turn for each event, in a first-in first-out order.

void remPostHook(SxDaemon \*devil)

Removes the daemon object **devil**, previously added by a call to `addPostHook()`, from the list of triggers.

void setWaitMode(InsertMode mode)

Changes the insertion policy of the thread waiting list. This method does not reorder the current waiting list contents, but rather sets the new mode for subsequent updates. mode can be one of the following:

- Ⓜ FIFO ensures that the oldest pending thread is always considered first.
- Ⓜ LIFO ensures that the latest pending thread is always considered first.

- ⑩ PRUP (or PRUPFF) ensures that the thread having the lowest priority is always considered first. When two threads have the same priority, the oldest pending one is priority.
- ⑩ PRUPLF ensures that the thread having the lowest priority is always considered first. When two threads have the same priority, the latest pending one is priority.
- ⑩ PRDN (or PRDNFF) ensures that the thread having the highest priority is always considered first. When two threads have the same priority, the oldest pending one is priority.
- ⑩ PRDNLF ensures that the thread having the highest priority is always considered first. When two threads have the same priority, the latest pending one is priority.

`InsertMode getWaitMode() const`

Returns the current insertion mode in the thread waiting list.

`virtual void alert()`

Resumes all threads currently waiting for the condition to be signaled. As a result of this operation, the current object's waiting list is emptied.

`virtual void forget(SxThread *thread)`

Removes the simulation **thread** from the object's waiting list.

`virtual void pend()`

Makes the current thread pend on the object. If the object's state is ON or POSTED, indicating that the condition is met, the method returns immediately to the caller. Otherwise, the current thread is linked to the object's waiting list, and suspended through a call to `SxThread::pend(SxSynchro *)`.

## SxTcp (extends SxListener)

### PURPOSE

This class implements a communication object above the *socket* interface, managing a bi-directional TCP/IP channel. This class is first candidate for providing the communication channel needed to support the monitoring facility through the SxMonitor object.

SxTcp is a subclass of the SxListener superclass, which allows a thread to wait asynchronously for input events on the TCP/IP channel, through the callout facility provided by the SIMEX thread scheduler.

Messages sent and received through an SxTcp instance are made of a message type identifier followed by an optional block of dynamically sized, unstructured data.

### CONSTRUCTORS

SxTcp()

Creates a client or server TCP/IP channel.

In client mode, the user subsequently issues a SxTcp::connect() request to the object to establish the connection with the server. In server mode, the user issues a SxTcp::bind() request to bind the object to a valid TCP port, followed by a call to SxTcp::accept() in order to wait for a client to connect to.

SxTcp(int s)

Creates a communication object from an already connected socket whose descriptor is given by s.

### DESTRUCTOR

virtual ~SxTcp()

Closes the socket channel associated with the object, unless the socket descriptor was obtained from the user. In all cases, all internal buffers held by the object are freed.

int getHandle() const

Returns the socket descriptor associated with the object.

int connect(const char \*host, int port)

Obtains a connection-oriented socket descriptor from the host operating system, then connects the client socket to the server running at **host** and listening to **port**. If **host** is a NULL pointer, the local host name is used.

This method asks for the *Nagle* algorithm to be disabled for the connection (i.e. TCP\_NODELAY, no packet coalescence). Moreover, the obtained socket descriptor is automatically registered as a monitored source at the **SxListener** level, through a call to **SxListener::addFildes()**.

Returns **SXTCP\_SUCCESS** on success, **SXTCP\_LINKDOWN** otherwise.

**int bind(int port =0)**

Obtains a connection-oriented socket descriptor from the host operating system, then binds the master server socket to **port**. If **port** is zero, the operating system will be asked to choose it freely.

The method tells the operating system to accept connections from any hosts from any networks. It always tries to reuse local addresses by setting the **SO\_REUSEADDR** flag at the socket protocol level.

The port to which the socket is finally bound is returned to the caller on success. Otherwise, **SXTCP\_FAILURE** is returned.

**int accept(u\_long timeout)**

Waits for a client socket to connect to the current object. The current object must have been configured as a server socket, and bound to a valid TCP port using **SxTcp::bind()**. The incoming connection is awaited for **timeout** seconds before returning an error status (seconds is meant to be “real” wall clock units here, rather than simulated time). However, if **timeout** is zero, the method waits indefinitely for the connection.

On success, the socket descriptor is automatically registered as a monitored source at the **SxListener** level, through a call to **SxListener::addFildes()**.

**SXTCP\_SUCCESS** is returned on success. Otherwise, **SXTCP\_LINKDOWN** is returned on protocol error, and **SXTCP\_WOULDBLOCK** is returned whenever the timeout expires before a connection is established.

**int send(int mid, const void \*mbuf =0, int nbytes)**

Sends a message of type **mid** and length **nbytes**, whose data starts from **mbuf**. The message only consists of the type information whenever **mbuf** is NULL or **nbytes** is zero or negative. **mid** must be a strictly positive integer not to conflict with error return values.

The transmission protocol enforced by the **SxTcp** object ensures that the message will be received as a whole by another peer **SxTcp** object which issues the corresponding call to **SxTcp::recv()** on the other end of the communication channel.

**nbytes** is returned on success. Otherwise, **SXTCP\_LINKDOWN** is returned on transmission error, such as a broken link detected with the peer.

**int recv(void \*\*mbufp, int \*ubytes)**

Receives the next available message from the connected peer. The starting address of the data buffer associated with the incoming message is written to **mbufp**, and its length is

written to **ubytes**. If no data buffer comes with the message, **\*mbufp** is set to NULL, and **\*ubytes** to zero.

Unless an error is detected, this method guarantees that message boundaries are kept between the sender and the receiver end-points.

Data buffers whose size fits in the object's internal receive buffer (statically allocated at object's creation) are returned directly from it. Messages longer than this size are transferred to a dynamically allocated buffer whose address is also returned to the caller. The size of the internal buffer is fixed by the constant **SXTCP\_MBUFSZ**, which is currently set to 4096 bytes. The dynamic buffer can be freed explicitly by a call to **dispose()**.

The message type identifier is returned on success. Otherwise, **SXTCP\_LINKDOWN** is returned on receive error.

**int poll(void \*\*mbufp, int \*ubytes)**

Attempts to receive the next available message from the connected peer, but does not block the caller if none is available. If a message is immediately available, this method acts like **SxTcp::recv()**.

Returns the incoming message type identifier on success. Otherwise, **SXTCP\_WOULDBLOCK** is returned, meaning that no message is immediately available for input.

**int dispose()**

Frees the memory associated with a current dynamic receive buffer (if any). The **SxTcp** implementation handles this deallocation from one call **recv()** to another, and at object's deletion, so there is usually no need to do it explicitly. However, this call is provided for unusual circumstances where forcing this release is needed.

## SxThread (extends SxTimed)

### PURPOSE

This class implements the SIMEX's thread object behavior. Threads are concurrent activities inside the simulation system having a private execution stack while sharing the global address space of the host application.

As timed objects, threads are prioritized objects scheduled according to their requested activation time. The list of runnable threads is called the *run chain*. The run chain is unique in the system, and is actually an instance of the **SxScheduler** class. Control of the CPU is always given to one SIMEX thread at any time during the simulator's life-time. The executing thread only changes whenever it issues a suspensive call, yielding control to the next runnable thread from the run chain. **SxThread::currentThread** is a pointer to the currently executing thread object (please note that *current* or *executing* are used indifferently to refer to the thread in control of the CPU).

A subtle distinction exists between the executive state of a thread, and its simulation state, which is of great importance. From the executive's perspective, a thread is owning the CPU or not. From the simulation's perspective, a thread can be in a running state, even if it does not currently own the CPU, provided it is a member of the run chain. In the latter case, the thread is said to be delayed. It is simply held until the executing thread yields control to another by calling a suspensive service, but the simulation date it was requested to run at is honoured, even if *real* hours of computation have taken place before the CPU is finally available to it.

Threads have four basic states:

- ⑩ IDLE denotes the unconditionally suspended state. The thread is removed from the run chain, and will not regain the CPU until it is explicitly resumed.
- ⑩ PENDED indicates the thread is waiting for the condition of a synchronization object to be met. Such object must be a subclass of the **SxSynchro** class. The thread is removed from the run chain, and will not regain the CPU until the condition is finally met or it is explicitly resumed.
- ⑩ PREEMPTED tells that the thread has been explicitly preempted by another one. The thread is removed from the run chain, and will not regain the CPU until it is explicitly resumed.
- ⑩ RUNNING denotes a running thread. The thread is whether currently executing or a member of the run chain.

### PUBLIC DATA MEMBERS

static SxThread \*currentThread

A pointer to the currently executing thread object.

static SxThread \*mainThread

A pointer to the main thread object. The main thread refers to the execution context which has created the **SxManager** global instance for the simulation process. This context is automatically re-hosted by a SIMEX thread object whose address is written to this variable. Unlike other threads, the main thread object has no allocated stack, but rather recycles the stack which was active at the time the manager was created.

### SxScheduler runChain

The thread scheduler object for the entire simulation system. The run chain is basically a list of timed objects the **SxThread** class manages. It links all threads being in the RUNNING state in a timely order, but the executing one. Threads having identical activation dates are ordered using their internal priority values (higher priority values first). Priority groups are managed on a FIFO basis.

### int fConservative

A flag indicating whether the thread manager runs in conservative mode. Please refer to the documentation of **setMtMode()** for more information on this topic.

## CONSTRUCTOR

**SxThread(const char \*name, int pflags =0, int stackSize =32768)**

Creates a thread object. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

**pflags** is a set of protocol flags such as defined by the **StProto** interface.

**stackSize** is the size (in bytes) of the execution stack the constructor will allocate for the new thread. A current minimum value of 8192 bytes is enforced by the constructor. A rather simple stack checking mechanism is active for each declared thread. An extra page guard of 1024 bytes is left at the bottom of the stack, and an overflow detection pattern is tested each time the thread invokes a suspensive call. This scheme should be sufficient for detecting most of the stack overflow problems; however, one should acknowledge the fact that stack memory is a scarce resource, and probably refrain from using unreasonably large automatic variables on behalf of SIMEX thread contexts.

The new thread inherits the priority of the current thread. Once started, the thread enters the RUNNING state and the **SxThread::body()** virtual method is invoked on behalf of its execution context. The child thread never preempts its creator at startup, ensuring the constructor is normally exited on behalf of the parent context before the new thread starts executing.

## DESTRUCTOR

**virtual ~SxThread()**

The *delete* operator and this destructor should never be called directly by the user for a thread object. Instead, thread deletion should rather be done by invoking the **cancel()** method.



The destructor first unlinks the thread from any synchronization objects' waiting lists it is member of. Finally, internal resources are freed, including the stack space.

## METHODS

`static void setMtMode(int fConservative)`

Changes the thread switching mode. If **fConservative** is true, a conservative way of switching thread contexts will be enforced by the thread manager. This mode ensures that the full context of a suspended thread is saved then restored when a switch occurs; it is slower than a light switch is, but safer under certain circumstances, such as running an executable image tweaked by an object code instrumenter (e.g. Purify™ et al.). Technically, the conservative mode differs from the faster one by calling the `sigsetjmp()/siglongjmp()` pair to save/restore thread contexts instead of `setjmp()/longjmp()`. Conversely, the fast-switch mode is enabled if **fConservative** is false.

`static void setGlobalTrace(int level)`

Set the current trace level of the **SxThread** methods to **level**. A value of 1 gives minimal trace information. A value of 2 adds detailed information to the output.

`void resume(SxSynchro *so)`

Resumes the thread object. **so** is an optional pointer to a synchronization object which may have motivated this call, usually because the condition awaited by the thread was finally met. **so** may be NULL if the action is independent from any specific synchronization object.

The actions taken by this method depend on the target thread's state:

- ⊗ if previously **PREEMPTED**, the thread is put in the run chain. If the thread was preempted while in the **RUNNING** state but not executing, the delay time remainder is renewed (see `preempt()`).
- ⊗ if previously **RUNNING** and not executing (i.e. delayed), the thread activation time is set to the current value of **SxClock** (i.e. "now"), and the run chain is reordered to reflect the change.
- ⊗ if previously **IDLE** or **PENDING**, the thread is inserted into the run chain with an activation time set to the current value of **SxClock** (i.e. "now").

As a result of this call, the target thread enters (or remains in) the **RUNNING** state. Resuming the executing thread leads to a null effect.

This method *does not preempt the executing thread* in any case.

`virtual void resume()`

Invokes `resume(NULL)`.

`void prioritize(int incr)`

Adds **incr** to the base thread priority value and sets the resulting priority to the target thread. The base priority is set to 100 for the current implementation. Increments which would lower the priority value to a negative number are silently rejected.

This method reorders the run chain to reflect the change, *but does not preempt the executing thread* in any case.

`void renice(int incr)`

Invokes `prioritize()` with the priority increment value **incr**, then attempts to preempt the executing thread if a priority thread is leading the run chain as a result of the operation.

`unsigned getStackSize() const`

Returns the thread's execution stack size (in bytes). As a special case, this method always returns 0 when invoked for the main thread.

`caddr_t getStackTop()`

Returns the thread's execution stack top address. As a special case, this method always returns NULL when invoked for the main thread.

`int onStackOverflow() const`

Returns a boolean value telling whether the target thread has overwritten its overflow detection pattern.

`int onStack(caddr_t addr) const`

Returns a boolean value telling whether **addr** belongs to the target thread's stack address space.

`void immediateResume(SxSynchro *so)`

Resumes immediately the thread object. **so** is an optional pointer to a synchronization object which may have motivated this call, usually because the condition awaited by the thread was finally met. **so** may be NULL if the action is independent from any specific synchronization object.

The executing thread is put at the front of the run chain, and CPU control is immediately relinquished to the target thread.

As a result of this call, the target thread enters (or remains in) the RUNNING state. Resuming the executing thread leads to a null effect.

`SxSynchro *waitUntil (SxSynchro *so, lTime timeout)`

Makes the target thread wait for the condition expressed by the synchronization object **so** to be met. The amount of time the thread can wait for the condition to be met is specified by **timeout**. If **timeout** equals ZEROTIME, the condition must be immediately satisfied for the call to succeed.

A thread switch may occur if the executing thread needs to wait for the condition to be met.

This calls returns either **so** if the condition is met within the allotted time, or NULL otherwise.

SxSynchro \*waitUntil (SxSynchro \*so)

Invokes waitUntil(**so**,ZEROTIME).

SxSynchro \*waitOrUntil (SxSynchroGroup \*sog, lTime timeout)

Makes the target thread wait for at least one condition among the group of synchronization objects **sog** to be met (disjunctive wait). The amount of time the thread can wait for a condition to be met is specified by **timeout**. If **timeout** equals ZEROTIME, the condition must be immediately satisfied for the call to succeed.

A thread switch may occur if the executing thread needs to wait for the condition to be met.

This calls returns a pointer to the first synchronization object from the group whose condition is met within the allotted time. Otherwise, NULL is returned indicating a timeout condition.

SxSynchro \*waitOr(SxSynchroGroup \*so)

Makes the target thread indefinitely wait for at least one condition among the group of synchronization objects **sog** to be met (disjunctive wait).

A thread switch may occur if the executing thread needs to wait for the condition to be met.

This calls returns a pointer to the first synchronization object from the group whose condition is met.

int xtry(SxThreadContext& buf)

Saves the target thread context information to **buf**. This call performs like setjmp() or sigsetjmp() depending on the current thread switching mode (see setMtMode() for more on this topic).

The saved context should be restored using xraise().

void xraise(SxThreadContext& buf)

Restores the target thread context using the information stored into **buf**, which must have been previously set by a call to xtry(). This call performs like a longjmp() or siglongjmp() depending on the thread switching mode which was in effect at the moment the context was saved.

If the target thread is currently executing, the stack frame is immediately unwind. Otherwise, the target thread will later resume execution using the restored context information.

virtual void cancel()

Cancels the target thread. An internal thread named “the undertaker” is immediately resumed to take in charge the thread deletion. Once resumed, the undertaker actually

deletes the canceled thread on behalf of its own context, which means that threads can cancel themselves safely.

After the effective deletion has taken place, the CPU control is gained by the thread leading the run chain.

Invoking `cancel()` is the required way of deleting a thread from the simulation executive. No other mean should be used, such as using the *delete* operator directly.

**virtual void pend(SxSynchro \*so)**

Makes the target thread wait the synchronization object **so** to be signaled. **so** may be NULL if the wait is unconditional.

The target thread is removed from the run chain, then enters the PENDING state. A thread switch occurs if the target thread is currently executing. Making a non-RUNNING thread pend leads to a null effect.

This method is usually called from synchronization objects as they put to sleep threads waiting for their condition to be met.

**virtual void activate()**

Makes the executing thread yield the CPU control to the target thread. The run chain is not altered by this call, but the global simulation clock variable (i.e. **SxClock**) and the executing thread pointer (i.e. **SxThread::currentThread**) are. The clock variable value bumps to the target thread activation time value, and the executing thread pointer is set to **this**.

This method is used by the executive's scheduler and should not be invoked directly by the user, unless low level SIMEX kernel code is involved.

**virtual void preempt()**

Preempts the target thread which must be in a RUNNING state. The thread is removed from the run chain until it is explicitly resumed, and enters the PREEMPTED state.

If the target thread is not executing, the delay remainder of the preempted thread is computed by subtracting the thread's next activation time stamp from the simulation clock current value (i.e. **SxClock**). This remainder will be renewed when the thread is resumed.

A thread switch occurs if the target thread is currently executing.

**virtual void delay(ITime dt)**

Delays the target thread which must be already in a RUNNING state. The delay period is given by **dt** which must be a positive or null internal time value. The next activation time is computed for the target thread and the run chain is reordered to reflect the change. The target thread is always inserted at the end of its priority group (i.e. round-robin effect) after the last thread having an earlier activation time and a higher priority.

Delaying a thread must not be confused with suspending it. The delayed thread remains runnable and does not leave the RUNNING state while it is (re-)inserted the run chain.

The major effect of delaying a thread is making the simulation clock *bump* to its scheduled activation time value when it is elected to regain the CPU.

A thread switch occurs if the target thread does not lead the run chain as a result of the operation.

`virtual void suspend()`

Suspends the target thread. If the thread was runnable, it is removed from the run chain until it is explicitly resumed. The target thread enters the IDLE state as a result of the suspension.

The suspension is not cumulative with the PENDING state. Therefore, a pending thread which gets suspended remains in the synchronization object's waiting list, and thus, is resumed when the condition is met. Only the thread object's intermediate state transitions will testify of such behavior (i.e. PENDING – IDLE – RUNNING instead of PENDING – RUNNING).

A thread switch occurs if the target thread is currently executing.

`virtual void timeout(SxTimer *timer)`

This callback method is invoked when a simulation timer (i.e. `SxTimer`) armed for the target thread has elapsed. The default action of this method is to resume the target thread.

It is safe to destroy the elapsed timer object on behalf of the `timeout()` callback if needed.

## SxTimed (extends SxObject)

### PURPOSE

This pure class is the superclass of objects having a behavior related to simulated time, such as threads (**SxThread**) and sources (**SxSource**). Each object inheriting this class is managed by a scheduler object (**SxScheduler**) which orders its activation in a timely manner.

A raw timed object has basically two states:

- Ⓢ IDLE if the object is not a current member of the scheduler's activation list.
- Ⓢ RUNNING otherwise, denoting a runnable object.

**SxTimed** class should be rarely inherited directly by user-level classes, but is rather used by low-level SIMEX kernel code.

### CONSTRUCTOR

**SxTimed**(const char \*name, **SxScheduler** \*sched, int pflags)

Creates a timed object. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

**sched** is the address of a valid scheduler object which will control the object activation during its life-time. A well-known scheduler object is the thread manager's run chain (see **SxThread**).

**pflags** is a set of protocol flags such as defined by the **StProto** interface.

### METHODS

**lTime** getTime() const

Returns the timed object's next activation time.

void setTime(**lTime** t)

Sets the timed object's next activation time to **t**. This method does not reorder the associated scheduler's activation list.

void insert(**lTime** t)

Invokes delay(**t**).

virtual void suspend()

Removes the timed object from the associated scheduler's activation list. The object enters the IDLE state as a result of the operation.

virtual void resume()

Inserts the timed object into the associated scheduler's list with an activation time equal to the current simulation clock value (i.e. "now").

`virtual void activate() =0`

This pure virtual method must be implemented by subclasses to take the necessary steps to make the target object executing. This method is called by the associated scheduler when the object's activation time has come.

`virtual void delay(ITime t)`

Sets the timed object's activation time to the current value of the simulation clock (i.e. `SxClock`) incremented by `t`. The increment must be a positive or null internal time value.

The target object must be the currently active timed object at the scheduler's level, otherwise, unexpected results may occur.

# SxCallout

## PURPOSE

Objects from the pure **SxCallout** class provide a simple mean of having callback routines executed all along the simulation process, whatever thread is currently in control. This functionality should be used to perform fast low-level tasks which need to run on a permanent basis in the background, but cannot be devoluted to a given simulation thread in particular.

Callouts are managed by the instance of the **SxScheduler** class they are attached to. Technically, they get fired each time the scheduler is asked to switch control to the next activable object (i.e. **SxScheduler::schedule()**).

A typical application of the callouts is the data channel listener (i.e. **SxListener**), which checks for available input for client threads on a set of file descriptors. Listeners are callouts attached to the thread manager's run chain.

## PROTECTED DATA MEMBERS

**SxThread \*boundThread**

The address of the thread object passed when constructing the current object.

## CONSTRUCTOR

**SxCallout(SxThread \*boundThread =0)**

Creates a callout object, optionally bound to the thread whose address is **boundThread**. This pointer is not directly used at the **SxCallout** superclass level, but is only maintained as a convenience for its subclasses. One usually sets this pointer to the address of a thread related to the callout action, if any. **boundThread** may be NULL if unused.

## METHODS

**virtual void process()**

This callback method is invoked by the scheduler object to which the current callout object is attached. The scheduler ensures that this callback is fired often and regularly, all along with the simulation process.



## SxListener (extends SxCallout)

### PURPOSE

This class implements a file descriptor monitoring object, used by threads to detect input conditions pending on input channel synchronously or asynchronously, while other threads are still allowed to run concurrently.

The simulation executive implements multi-threading on behalf of co-routines, not native threads from the host operating system. This is relevant with the idea of time independence which is needed to have an event-driven scheduling kernel which guarantees reproducible behavior of threads accross simulation sessions. The simulation threads (i.e. **SxThread** instances) are executed in a serialized order from the host operating system's prospective. Thus, it becomes quite obvious that issuing indeterminately blocking system calls to the host operating system on behalf of a simulation thread is a bad idea, unless one doesn't mind that a given thread blocks the whole simulation process while waiting for an event/resource which is external to the simulator itself. The listener object has been designed to circumvent this constraint.

A listener monitors a given set of file descriptors for input, resuming a target thread each time the condition is met. Once created, a listener should be attached to the thread manager's run chain using **SxScheduler::addCallout()**.

### CONSTRUCTORS

**SxListener(int fd ==-1)**

Creates a listener to monitor the file descriptor **fd**. More descriptors can be added subsequently by calling **addFildes()**. With a negative argument, this constructor creates an idle listener.

**SxListener(fd\_set \*waitSet)**

Creates a listener to monitor the set of file descriptors **waitSet**. The initial wait set can be updated using **addFiles()** and **removeFildes()**.

### METHODS

**const fd\_set& getWaitSet() const**

Returns the current wait set, composed of the monitored file descriptors.

**int getWaitCount() const**

Returns the number of file descriptors monitored in the current wait set.

**const fd\_set& getReadySet() const**

Returns the set of file descriptors which are marked as having pending input since the last check. The current ready set is also altered by **SxListener::poll()** invoked with a null ready set pointer. A client thread resumed by the listener usually calls this method to retrieve the readied file descriptor(s).

**void addFildes(int fd)**

Adds **fd** to the set of monitored file descriptors. Calling this method for an already set descriptor is harmless.

**void removeFildes(int fd)**

Removes **fd** from the set of monitored file descriptors. Calling this method for an unset descriptor is harmless.

**int poll(fd\_set \*readySet, struct timeval \*tv)**

Attempts to check for available input on the monitored file descriptors immediately. This call simply invokes the synchronous i/o multiplexing service (i.e. **select(2)**) from the host operating system with the current wait set as input. The resulting set is stored in **readySet**. If **readySet** is NULL, the internal ready set is used.

**tv** is used to specify the allotted amount of time for the condition to be met before **poll()** returns with a zero value. If **tv** is NULL, the wait is infinite. If **tv** is valid, and both *tv\_usec* and *tv\_sec* are zero, **poll()** immediately returns with the available status. Otherwise, the wait is limited to the amount of time given by this parameter. It should be noted that in such case, the host time representation is applied, not the simulated time.

This method returns the number of ready file descriptors in the ready set. If **readySet** was NULL, the returned set can be retrieved by a call to **getReadySet()**.

**int poll(fd\_set \*readySet)**

Invokes **poll(fd\_set \*readySet, struct timeval \*tv)** with **tv** being a valid, zeroed time specification; this causes the method to return immediately with the available status.

# SxScheduler

## PURPOSE

The SIMEX scheduler is in charge of managing a set of runnable time-related objects, maintaining their respective activation order. Timed objects (i.e. **SxTimed**) are first ordered by scheduled activation date, then by decreasing priority. Objects with identical activation dates and priorities are scheduled according to the FIFO insertion order. At any time, the object having the highest priority among those having the earliest activation date is active.

The current time reference used by the **SxScheduler** class is the contents of the **SxClock** variable.

The thread manager's run chain is an instance of the **SxScheduler** class (see **SxThread**).

## CONSTRUCTOR

**SxScheduler()**

Creates a scheduler object. At this point, there are neither runnable nor active objects.

**void insert(SxTimed \*timed, lTime t)**

Inserts the runnable **object** in the scheduling list, with a relative delay of **t**. The absolute activation date of **object** is set to **SxClock + t**.

**void insert(SxTimed \*timed)**

Inserts the runnable **object** in the scheduling list, with an activation date equal to **SxClock** (i.e. "now").

**SxTimed \*exchange()**

Activates the next runnable object and returns its address. The previously active object is pushed back at front of the scheduling list. If there is no runnable object currently linked to the scheduler, a fatal error is raised.

A timed-object is activated by invoking its **activate()** callback method.

**virtual SxTimed \*schedule()**

Activates the next runnable object and returns its address. If there is no runnable object currently linked to the scheduler, a fatal error is raised.

A timed-object is activated by invoking its **activate()** callback method.

**int isActive(SxTimed \*object) const**

Returns a boolean value telling whether **object** is currently active for the scheduler. An object gets active whenever it is elected to run either by the **schedule()** or **exchange()** methods.

SxTimed \*getActiveObject()

Returns the address of the active object for the scheduler. NULL is returned before the first runnable object is inserted.

void addCallout(SxCallout \*callout, SxThread \*boundThread =0)

Adds **callout** to the list of callouts the scheduler will fire before each object activation. **boundThread** is an optional pointer to a thread object bound to the callout.

Please refer to the SxCallout documentation for more on this topic.

void removeCallout(SxCallout \*callout)

Removes **callout** from the list of active callouts. Calling this method for an unregistered – but valid – callout object is harmless.

Please refer to the SxCallout documentation for more on this topic.

## SxSlaveScheduler (extends SxScheduler)

### PURPOSE

This class implements a slave scheduler depending on a primary scheduler (SxScheduler) to activate the runnable timed-object it holds. This quite exotic object is mostly used by SIMEX source and timer managers.

SxSchedSlave(SxScheduler \*master =0, SxTimed \*manager)

Creates a slave scheduler. **master** is a pointer to the primary scheduler holding the **manager** object. Both pointers may be NULL at object creation. In such a case, the correct master and manager addresses should be set subsequently using **setMaster()** and **setManager()** before the first runnable object is scheduled.

void setMaster(SxScheduler \*sched)

Sets the primary scheduler address to **sched**.

void setManager(SxTimed \*manager)

Sets the activation manager address to **manager**.

virtual SxTimed \*schedule()

Activates the next runnable object and returns its address. If there is no runnable object currently linked to the scheduler, NULL is returned.

The elected object is activated indirectly, by scheduling its activation manager held by the primary scheduler at the object's activation date. As a special case, a suspended manager is simply resumed at the current date.

## SxTimer (extends SxTimed)

### PURPOSE

This class implements a timer object, allowing to perform specific actions at predefined simulation dates.

A timer has basically three active states:

- ④ IDLE denotes an idle timer object.
- ④ ARMED indicates that an expiration date currently exists for the timer.
- ④ DISARMED indicates that a request to disarm the previously armed timer has been issued.
- ④ EXPIRED denotes an expired timer.

The current time reference used by the timer manager is the contents of the global clock variable **SxClock**.

### CONSTRUCTOR

**SxTimer(const char \*name, ITime timeout, SxThread \*wthread =0, int pflags =0)**

Creates a simulation timer. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

The timer is immediately armed with an expiration date of **SxClock + timeout**.

**wthread** is the optional address of a bound thread, whose **timeout()** callback will be fired at timer expiration. If this pointer is NULL, the timer will simply enter the EXPIRED state.

**pflags** is a set of protocol flags such as defined by the **StProto** interface.

**SxTimer(const char \*name, SxThread \*wthread =0, int pflags =0)**

Creates a simulation timer. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

The timer is idle.

**wthread** is the optional address of a bound thread, whose **timeout()** callback will be fired at timer expiration. If this pointer is NULL, the timer will simply enter the EXPIRED state.

**pflags** is a set of protocol flags such as defined by the **StProto** interface.

**SxTimer(const char \*name =0, int pflags =0)**

Creates a simulation timer. **name** is a null-terminated character string identifying the new object which can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

The timer is idle and has no bound thread.

**pflags** is a set of protocol flags such as defined by the **StProto** interface.

**void set(ITime timeout)**

Arms the timer. The expiration date is set to **SxClock + timeout**. Re-arming an armed timer is perfectly legal; in such a case, the expiration date is simply set to the new value.

**void reset()**

If the timer is armed, disarms it. Otherwise, the IDLE state is entered.

**virtual void activate()**

This callback method is invoked by the timer manager when the current timer expires. Its default action is to make the timer enter the EXPIRED state, then invoke the **timeout()** callback method of any bound thread.

## SxTrigger (extends SxThread)

### PURPOSE

SxTrigger is the superclass of trigger subclasses. Triggers are threads driven by event sources to start a client handler. The handler is fired according to the underlying source, which may be periodic, exponential, and so on.

This class should not be instantiated “as is”, but it should be extended by specific trigger implementations.

### PROTECTED DATA MEMBERS

SxSource \*source

The address of the source object driving the trigger object.

void (\*handler)(void \*)

The address of the user-defined handler to call.

void \*clientData.

The user-provided cookie to pass to the handler.

SxQueue wakeupQ

The internal message queue pended by the trigger’s body which gets posted by the source object.

### CONSTRUCTOR

SxTrigger(void (\*handler)(void \*cookie), void \*cookie, unsigned stackSize =SXTRIGGER\_MINSTACKSZ)

Creates a trigger object. **handler** is the user-provided handler to call with the argument **cookie**. **stackSize** is passed to the SxThread constructor. Controlling the stack size may be useful if complex actions are to be performed by the handler.

virtual int isValid() const

Returns a boolean value indicating whether the trigger object has a valid source.

virtual void body()

Implements the main loop for the trigger object thread. The handler gets fired each time a new message is obtained from the internal queue bound to the driving source.



## SxPerTrigger (extends SxTrigger)

### PURPOSE

This class implements a periodic trigger. Objects from this class are driven by a periodic source (SxPerSource).

### CONSTRUCTOR

SxPerSource(void (\*handler)(void \*cookie), const char \*param, void \*cookie =0, int priority =0, unsigned stackSize =SXTRIGGER\_MINSTACKSZ)

Creates a periodic trigger. **handler** is the user–provided handler to call with the argument **cookie**.

**stackSize** is passed to the SxThread constructor. Controlling the stack size may be useful if complex actions are to be performed by the handler.

**param** is the source parameter string, which must be formatted as *[tMin–tMax/]period*, where *tMin* and *tMax* are the optional source activation bounds, and *period* is the trigger period.

Messages generated by the periodic source are given the **priority** value.

## SxExpTrigger (extends SxTrigger)

### PURPOSE

This class implements an exponential trigger. Objects from this class are driven by a periodic source (SxExpSource).

### CONSTRUCTOR

SxExpSource(void (\*handler)(void \*cookie), const char \*param, void \*cookie =0, int priority =0, unsigned stackSize =SXTRIGGER\_MINSTACKSZ)

Creates an exponential trigger. **handler** is the user-provided handler to call with the argument **cookie**.

**stackSize** is passed to the SxThread constructor. Controlling the stack size may be useful if complex actions are to be performed by the handler.

**param** is the source parameter string, which must be formatted as *[tMin–tMax/]mean*, where *tMin* and *tMax* are the optional source activation bounds, and *mean* is the source's mean.

Messages generated by the exponential source are given the **priority** value.

## SxFileTrigger (extends SxTrigger)

### PURPOSE

This class implements a file-based trigger. Objects from this class are driven by a file-based source (SxFileSource).

### CONSTRUCTOR

SxFileSource(void (\*handler)(void \*cookie), const char \*fileName, void \*cookie =0, int priority =0, unsigned stackSize =SXTRIGGER\_MINSTACKSZ)

Creates a file-based trigger. **handler** is the user-provided handler to call with the argument **cookie**.

**stackSize** is passed to the SxThread constructor. Controlling the stack size may be useful if complex actions are to be performed by the handler.

**fileName** is the path of the file containing the time specifications.

Messages generated by the file-based source are given the **priority** value.

## SxUniTrigger (extends SxTrigger)

### PURPOSE

This class implements a uniform trigger. Objects from this class are driven by a uniform source (SxPerSource).

### CONSTRUCTOR

SxUniSource(void (\*handler)(void \*cookie), const char \*param, void \*cookie =0, int priority =0, unsigned stackSize =SXTRIGGER\_MINSTACKSZ)

Creates a uniform trigger. **handler** is the user–provided handler to call with the argument **cookie**.

**stackSize** is passed to the SxThread constructor. Controlling the stack size may be useful if complex actions are to be performed by the handler.

**param** is the source parameter string, which must be formatted as  $[tMin-tMax/]dMin-dMax$ , where  $tMin$  and  $tMax$  are the optional source activation bounds, and  $dMin$  and  $dMax$  are the distribution bounds.

Messages generated by the uniform source are given the **priority** value.

## SxTimerTrigger (extends SxTrigger)

### PURPOSE

This class implements a one-shot timer trigger. As a special exception, objects from this class are not driven by a source.

Once expired, a timer trigger suspends itself.

### CONSTRUCTOR

SxTimerSource(void (\*handler)(void \*cookie), const char \*param, void \*cookie =0, unsigned stackSize =SXTRIGGER\_MINSTACKSZ)

Creates a timer trigger. **handler** is the user-provided handler to call with the argument **cookie**.

**stackSize** is passed to the SxThread constructor. Controlling the stack size may be useful if complex actions are to be performed by the handler.

**param** is the timer parameter string, which must be a valid time specification.