LunaSDL Reference Manual

Stefano Trettel

Version 0.2
2015-02-23

Table of Contents

Introduction

This manual describes the functions provided by LunaSDL. It is assumed that the reader is familiar with the Lua programming language. Familiarity with SDL may be of help, but is not essential to read this document. [This manual is written in AsciiDoc, rendered with AsciiDoctor and a CSS from the AsciiDoctor Stylesheet Factory. The PDF version is produced with AsciiDoctor-Pdf. The SDL diagrams are made with the yED Graph Editor and a custom palette of SDL symbols made with Inkscape.] [Introductive papers to SDL, a bit outdated but still very good, can be found in the Telektronikk journal, *Volume 4.2000 "Languages for Telecommunication Applications"*, Ed: Rolv Bræk.)]

What is LunaSDL

LunaSDL is a **Lua module for event-driven concurrent programming**, whose design follows the concurrency model of the **ITU-T Specification and Description Language (SDL)**.

LunaSDL turns Lua into an SDL dialect by extending it with the basic constructs for the implementation of SDL systems, which are systems composed of concurrent, reactive and intercommunicating entities referred to as **agents**. [LunaSDL is to be considered an *SDL dialect* in that it slightly deviates from the SDL standards and has some simplifications: it uses an all-to-all pid-based communication model, with a single signal queue instead of per-agent input queues (although one could argue that the per-agent queues ar just merged but still distinguishable by the destination pid...); it has no explicit definitions of channels and gates; in LunaSDL any agent (not only the system agent) may communicate with the environment; it uses Lua data types and has no built-in support for ADT and ASN.1; it has additional non-SDL constructs like priority outputs, time-triggered signals and remote synchronous functions with immediate return.]

What LunaSDL is not. It is not an editor or a validator of SDL specifications, and it is not a code generator from SDL specifications (it may, however, be used as target language for tools that do such things). Moreover, LunaSDL is not intended for parallel multithreading and multi-core programming, nor for hard real-time applications.

Getting and installing

The **official repository** of LunaSDL is on GitHub at the following link: **https://github.com/stetre/lunasdl** .

LunaSDL requires **Lua** version 5.2 or greater, and **LuaSocket**.

Since it is written in plain Lua, no compiling is needed.

LunaSDL has been tested on Linux (Fedora 21) and on OpenBSD (5.6). It may run on any other OS supported by Lua and LuaSocket, but this has not been tested.

To install LunaSDL, download the latest release and extract it somewhere on your system.

To use LunaSDL, make sure that the base directory containing the `lunasdl.lua` script is in Lua's *package.path* (for this purpose, there is an example `configure` shell script in the base directory.)

For example, on a GNU/Linux system, you may do this:

```
# ... download lunasdl-0.1.tar.gz ...
[ ]$ tar -zxpvf lunasdl-0.1.tar.gz
[ ]$ cd lunasdl-0.1
[lunasdl-0.1]$ . configure
```

Or, using wget:

```
[ ]$ wget https://github.com/stetre/lunasdl/archive/v0.1.tar.gz
[ ]$ tar -zxpvf v0.1.tar.gz
[ ]$ cd lunasdl-0.1
[lunasdl-0.1]$ . configure
```

Some basic examples can be found in the `examples/` directory and are described in the Examples section of this manual.

Common to all examples, and to LunaSDL applications in general, is the need to properly set the `LUNASDL_PATH` environment variable so that LunaSDL can find the scripts used by the application. For more details, see the Finding agent scripts section.

Module organization

The LunaSDL module is loaded using Lua's *require* function and returns a table containing the functions it provides (as usual with Lua modules). This manual assumes that such table is named ***sdl***, i.e. that it is loaded with:

```
sdl = require("lunasdl")
```

but nothing forbids the use of a different name.

As explained later in the manual, LunaSDL creates multiple Lua environments, one per SDL agent. All these environments share the same ***sdl*** table. In each agent's environment, LunaSDL defines also some special global variables that it uses to share information with the agent, and some other special variables for internal use (mostly packed in another table named ***sdl***_). To avoid conflicts with user-defined identifiers, a minimal naming convention is used. See the Special variables section for more details.

License

LunaSDL is released under the **MIT/X11 license** (same as Lua, and with the same only requirement to give proper credits to the original author). The copyright notice is in the LICENSE file in the base directory of the official repository on GitHub.

Overview

LunaSDL applications

A **LunaSDL application** is any application written in Lua that relies on LunaSDL to implement a system of concurrent, reactive and intercommunicating entities. In the SDL jargon, and thus in LunaSDL, such entities are called **agents**. You can think of them as of cooperative threads of execution, akin to Lua's native coroutines.

In its most natural form, a LunaSDL application is composed of:

- a **main script**, that loads and configures the LunaSDL module, creates the first agent, and then enters an event loop, and
- one or more **agent scripts**, each of them defining the behavior of an agent (more precisely, of a *type* of agent).

Agents are created with the functions of the *sdl.create family*, and each of them has its script executed in a dedicated Lua environment, which provides separation of its namespace from those of other agents running in the same application.

Communication

Agents communicate mainly by sending **signals** to each other, and may create and set **timers**, whose expiries are also notified to agents by means of signals.

Agents may also communicate with the outside world with respect to the system (i.e. the *environment* [In SDL, the outside world with respect to a system is called the *environment*. Not to be confused with the *Lua environment*.]) by means of **file descriptor objects** like, for example, sockets. Multiple file descriptor objects can be used concurrently.

Reactivity and concurrency

Agent scripts define the behavior of agents in terms of state machines, whose transitions are triggered by events represented by the reception of signals.

The interleaved execution of agents is controlled by an **event loop** that dispatches signals one at a time.

Every time the event loop dispatches a signal, the agent which is the destination of the signal becomes the **current agent**, executes some code depending on its state and on the signal, and then returns to the event loop (the model of concurrency is *cooperative*). The event loop then dispatches the next signal, if any is scheduled, otherwise it waits for stimuli, which may be the expiry of timers or file descriptors that have become ready for read or write operations.

Agents hierarchy

With the exception of the first agent, which is created at startup, every agent in a LunaSDL application is created by another already existing agent, and is linked to it in a *child-parent* relationship. Thus, agents are naturally organized in a hierarchical tree, whose root is the first agent and is called the **system agent**.

There are three *kinds* of agents: **system**, **block** and **process**. Only one agent (the first created) is of the kind *system*, while the others may be of the kinds *block* or *process*, the difference between the two being that a *block* represents a container of agents (with an associated domain name) while a *process* does not.

Every SDL agent is also directly contained in an SDL agent of the kind *block* or in the *system agent* (which is also a block, although a special one). The block an agent is contained in is its parent, if this is a block, or the same block the parent is contained in, otherwise.

> **NOTE** Conventionally, the system agent's parent is itself, and it is contained in itself.

Agents identification

Agents are identified by their **process identifier (pid)**, which is unique within a LunaSDL application. The pid is an integer value automatically assigned to an agent at its creation, and is used as address when sending signals. The pid value *0 (zero)* always identifies the system agent.

Each agent is also assigned, by its parent, an **agent name** (a string) which is required to be unique within the block the agent is contained in, but may be reused in different blocks.

The agent name may encode information such as the agent's 'type' and 'instance' (but this is up to the application designer) and provides a convenient way to identify agents in reusable agent scripts without relying on the pid values being always the same from application to application (which is generally not true, being pids dynamically assigned).

Functions are provided to resolve agent names into pids and viceversa.

The main script

A typical main script of a LunaSDL application looks like in the example below. The application is executed like any other regular Lua script, for example using the standalone Lua interpreter:

```
~$ lua main.lua
```

Main script example:

```lua
-- main.lua
local sdl = require("lunasdl")

-- ... get arguments from command line or from elsewhere...
-- ... optionally configure LunaSDL ...

-- Open the system log file
sdl.logopen("mylogfile.log")


-- Create the system agent as defined by the agent script "mysystem.lua",
-- give it the agent name "System", and enter the event loop:
assert(sdl.createsystem("System","mysystem"))
```

The script loads the LunaSDL module, optionally configures it, and then it creates the first agent (i.e. the *system agent*), which in turn may or may not create other agents, depending on the application.

The *sdl.createsystem* call, besides creating the system agent, enters the event loop and returns only when the system stops.

Agent scripts

An agent script defines the behavior of an agent in terms of a state machine, whose transitions are triggered by the arrival of input signals. It may look like in the example shown below, in which the bottom part defines the transitions of the state machine, associating Lua functions to combinations of states and input signals names, and the top part implements those Lua functions.

Agent script (incomplete) example:

```
-- agent.lua

local T = sdl.timer(30,"T_EXPIRED")
local somevariable = 0

function Start()
  -- .. 'start transition' here ..
  sdl.nextstate("Waiting")
end

function Waiting_ConReq()
  -- ... transition: received CONREQ signal in Waiting state ...
  sdl.send({ "CONACK" }, sender_)
  sdl.set(T)
  sdl.nextstate("Connecting")
end

function Connecting_TExpired()
  -- ... transition: received T_EXPIRED signal in Connecting state ...
  sdl.send({ "FAILURE" }, parent_)
  sdl.stop()
end

-- ... cut ...

-- the state machine:
sdl.start(Start) -- sets the 'start transition
--             state, input signal, transition function
sdl.transition("Waiting","CONREQ",Waiting_ConReq)
sdl.transition("Connecting","CONCNF",Connecting_ConCnf)
sdl.transition("Connecting","T_EXPIRED",Connecting_TExpired)
sdl.transition("Connected","DATA",Connected_Data)
sdl.transition("Any","STOP",Any_Stop)
sdl.default("Any") -- the default state ('asterisk state')
```

State machine functions

In order to define state machines, LunaSDL provides the functions described hereafter. More details can be found in the subsections that follow.

- **sdl.start** (*func*)

  Sets the function *func* as the agent's *start transition*.

- **sdl.transition** (*state*, *signame*, *func*)

  Sets the function *func* as the transition to be executed when a signal named *signame* (a string) arrives with the agent being in the *state* state (also, a string).

  The value '*' (asterisk) may be passed as argument for the *signame* parameter, meaning *'any signal name for which an explicit transition has not been set for this state'*.

- **sdl.default** (*state*)

  Sets *state* (a string) as the agent's default state. The default state is optional, and can be used to define transitions for signals which are not catched with *sdl.transition* in the state the agent is in when they arrive.

- **sdl.nextstate** (*state*)

  To be used within transitions, changes the current state of the agent to *state*.

  If the state actually changes, i.e. if the agent was not already in that state, this function also re-schedules any input signal previously saved with *sdl.save*. Otherwise calling this function is superfluous but harmless.

- **sdl.stop** ([*atstopfunc*])

  To be used within transitions, gracefully stops the agent's state machine and determines the termination of the agent itself.

  The optional *atstopfunc* argument (a function) is a finalizer to be called when the agent actually terminates.

- **sdl.kill** ([*pid*])

  To be used within transitions, ungracefully terminates the agent identified by *pid* and all its descendants, without passing through the *stopping condition*. The calling agent must be an ascendant of the agent to be killed, or the agent itself. The *pid* argument defaults to ***self_***.

Transitions

A **transition** is a function implemented in the agent script, set by means of the above functions as the code to be executed at the arrival of an input signal (or, in the case of the start transition, at the creation of the agent).

When executed, a transition performs some task depending on the input signal and on the state the agent is in (e.g., it may process input data, set timers, send output signals, and possibly change the agent's state), and then it returns.

To return (and as soon as possible) is the main requirement for a transition, because the concurrency model of SDL is *cooperative*: a transition must not be blocking, that is, it must not contain infinite loops or calls to blocking functions.

Each agent has a dedicated Lua environment where the agent script and the transitions are executed, so global functions and variables of an agent do not collide with those of other agents (they are *global* only in its dedicated environment).

The start transition

The **start transition** is the first transition of the agent's state machine and is automatically executed right after the agent has been created.

When an agent is created, LunaSDL first initializes its dedicated Lua environment, then it loads and executes the agent script, and finally it calls the start transition function that was set by the script with *sdl.start*. The function receives as arguments those (if any) that were in turn passed to the create function in its variable arguments part.

The start transition must contain a *sdl.nextstate* call to set the first state entered by the agent, or end with a *sdl.stop* call to terminate the agent without entering any state.

Once the start transition is executed, the control returns to the parent agent. The newly created agent will then be awakened again whenever an input signal addressed to it is dispatched by the event loop, causing the execution of the proper transition which is determined as described below.

Receiving input signals

When an input signal is dispatched to an agent, LunaSDL determines the transition triggered by it, then it executes it in the agent's dedicated Lua environment.

Assuming the signal name is *signame*, and the agent's current state is *state*, the triggered transition is the first found by LunaSDL in the list that follows and in the exposed order:

1. the transition explicitly defined in *state* for *signame*, or
2. the *asterisk* transition defined in *state*, or
3. the transition explicitly defined in the default state for *signame*, or
4. the *asterisk* transition defined in the default state, or
5. the *empty transition*, which implicitly consumes the signal by doing nothing and leaving the agent in the state it was before the arrival of the signal.

The signal contents and other relevant information such as the sender's pid are passed to the agent by means of some special variables that are properly set by LunaSDL before executing the transition (the special variables are those prescribed by the SDL standards, with a few additions).

Stopping an agent

A *sdl.stop* call in a transition causes the end of the agent's state machine and puts the agent in a **stopping condition** which preludes its termination.

The agent remains in the stopping condition until all its children have terminated, then it terminates too. While in the stopping condition, it will not receive any input signal (its state machine has ended), but it will remain available for remote functions calls.

If an *atstopfunc* finalizer is passed when invoking *sdl.stop*, it is executed right before the agent actually terminates and after all his children have terminated.

If the stopping agent is the system agent, its termination causes also the termination of the event loop and the return from the *sdl.createsystem* function call. Notice that this happens when the system agent actually terminates, i.e. when all its descendants have already terminated (and thus it is the last agent left in the system).

Creating agents

Agents are created using the functions that follow, all of them accepting the same arguments, namely:

name

the agent's name (a string), which is required to be unique in the block the agent is contained in. If *name=nil*, then the default name *"agent<pid>"* is automatically assigned to the agent (for example, the default name for the system agent is *"agent0"*).

script

the agent script defining the agent (see also *Finding agent scripts* below).

...

optional arguments to be passed to the start transition function set by the script.

- **sdl.createsystem** (*name* , *script*, ...)
  true, *returnvalues*

  Creates the *system agent* and enters the event loop. Any other agent must be created by the system agent itself or by one of its descendants using the *sdl.createblock* or the *sdl.create* functions described below.

  This function exits the loop and returns only when the system agent terminates - which can happen only after all the other agents have terminated too - or if an error occurs, in which case it returns *nil* and an error message (notice that this means that it does not call Lua's *error* function as most LunaSDL functions do).

  On success, the function returns *true*, possibly followed by a list of values (if any were set, during the execution of the system, using the *sdl.systemreturn* function).

- **sdl.createblock** (*name* , *script*, ...)
  pid

  Creates an agent of the SDL kind *block* and returns its *pid*. A block agent can be created only by other block agents, including the system agent (which is also a block, although a special one).

- **sdl.create** (*name* , *script*, ...)
  pid

  Creates an agent of the SDL kind *process* and returns its *pid*. A process agent can be created either

by block agents (including the system agent) or by other processes.

System return values

- **sdl.systemreturn** (...)

  Sets the values to be returned by the *sdl.createsystem* function when the system stops gracefully.

Finding agent scripts

To find agent scripts, the functions of the *sdl.create* family use the same mechanism that Lua uses to find modules and packages.

More precisely, the *script* argument is resolved by invoking the standard Lua *package.searchpath* function, passing it the templates contained in the variable ***sdl.path*** as the argument for the *path* parameter.

The *sdl.path* variable defaults to `"?;?.lua"` so that if, for example, an agent is created like so:

```
pid = sdl.create("myagentname","mydir.myscript")
```

then LunaSDL searches for a file named `"mydir/myscript"` or `mydir/myscript.lua"`, in this order.

The default *sdl.path* can be overridden by setting the `LUNASDL_PATH` environment variable with the desired path templates (in the same way one sets the standard `LUA_PATH` variable to override the default *package.path*).

Signals

Format of signals

Signals exchanged between agents in LunaSDL, as well as signals generated by timers, are Lua tables whose first array element (i.e. the element with numeric key=*1*) must be a string and denotes the **signal name**. For example:

```
mysignal = { "MYSIGNAL", "hello", 1, 2, 3, self_, true }
-- "MYSIGNAL" is the signal name; the fields values follow
```

The signal in the example above has only array fields, but the record part of the table can also be used.

Besides expecting the signal name in the first array position, LunaSDL makes no other assumptions regarding the format and meaning of signals and of their contents: it just delivers them between agents. Note that signals are **sent by reference**.

The functions that can be used by agents to send, save and possibly re-schedule saved signals are described in the following subsections.

Sending signals

- **sdl.send** (*signal*, *dstpid*, [, *priority*])
  now

  Sends the SDL *signal* to the agent identified by *dstpid*, with the specified *priority*, and returns the current time.

  The *priority* argument, if present, should be an integer between *1* (high) and *N* (low), where *N* is the number of priority levels optionally configured at startup.

  If *priority* is not specified, or if it is greater than *N*, the signal is sent without priority (which means lowest).

  NOTE

  The SDL-experienced reader may have noticed that LunaSDL has *priority outputs*, while standard SDL has *priority inputs*. The following function somehow straightens things by allowing the receiver agent to specify the priorities of input signals, and override the priorities specified by sender agents.

- **sdl.priorityinput** (*signame*, [*priority*])

Configures the *priority* for input signals named *signame*.

This function is (optionally) used by an agent to impose priorities for input signals. If it is called with a non-*nil priority* argument, then signals named *signame* sent to this agent will be sent with the specified *priority*, no matter the priority specified by the sender.

If *priority* is *nil*, the desired priority is reset and the priority specified by the sender applies.

The *sdl.priorityinput* function can be used by an agent at any time, but only signals sent after it is called are affected by it, i.e. signals already sent but not yet dispatched are not affected (so, beware of subtle differences with the standard SDL priority input construct).

Signals generated by timers, time-triggered signals and re-scheduled signals are not affected either.

Time-triggered signals

- **sdl.sendat** (*signal*, *dstpid*, *at* [, *maxdelay*])

Schedules the SDL *signal* to be sent to the agent identified by *dstpid* at the point in time specified by *at*.

The optional *maxdelay* argument is the maximum delay, in seconds after *at*, after which the signal must be considered stale, and discarded. A *nil* value for *maxdelay* means *infinity* (i.e. the signal never expires).

When a signal is sent with this function, LunaSDL retains it in an internal queue and only when the time specified by the *at* argument arrives, it sends it to the destination agent (with maximum priority). If, for some reason, the signal can not be delivered before the point in time given by *at* + *maxdelay*, LunaSDL silently discards it.

NOTE

This construct is not part of the SDL standard. It is instead inspired by a proposal contained in the paper "Real-time signaling in SDL", by M. Krämer, T. Braun, D. Christmann and R. Gotzhein, published in *SDL'11 Proceedings of the 15th international conference on Integrating System and Software Modeling*, Springer-Verlag, 2011.

Saving and re-scheduling signals

- **sdl.save** ( )

To be used within transitions triggered by input signals, saves the current input signal (i.e. the content of the ***signal_*** special variable) in the agent's *saved queue*, for later re-scheduling.

Saved signals are automatically re-scheduled when the agent changes its state with a *sdl.nextstate* call, but can be re-scheduled explicitly at any time by means of the *sdl.restore* function described

below.

In both cases, saved signals are re-scheduled without priority (even if they were originally sent with priority) and in the same order they were saved.

- **sdl.restore** ( )

  Explicitly re-schedules all the signals saved in the agent's *saved queue*.

Timers

SDL agents can create and manage timers using the functions described here.

- **sdl.timer** (*duration*, *signame*)
  tid

  Creates an SDL timer and returns a unique **timer identifier (tid)** to be used in subsequent operations. The timer is *owned* by the agent that created it. Timer functions can be invoked on a specific timer only by its owner agent or by procedures that act on its behalf.

  The *duration* parameter is the timer's default timeout in seconds, and the *signame* parameter (a string) is the name of the SDL signal sent by LunaSDL to the owner agent when the timer expires.

  NOTE A timer can be created by an agent only before the execution of its start transition. Timers can not be created in state transitions, including the start transition, nor within procedures. Procedures can, however, use timers owned by their calling agent.

- **sdl.modify** (*tid*, *duration* [, *signame*])

  Modifies the default duration and/or the signal name of the timer identified by *tid* (to modify only the signal name, *nil* must be passed explicitly as argument for *duration*).

  The *duration* and *signame* parameters have the same meaning as for the *sdl.timer* function above.

  If the timer is active, this function also resets (stops) it.

- **sdl.set** (*tid* [, *at*])
  now

  Sets (starts) the timer identified by *tid* so to expire at the point in time given by *at*, and returns the current time.

  The *at* parameter is optional and defaults to *sdl.now()* + *duration*, where *duration* is the default duration specified for the timer.

  When the timer expires, an SDL signal is sent to the owner agent. Such signal contains only the signal name specified when the timer was created or later modified.

- **sdl.reset** (*tid*)
  now

  Resets (stops) the timer identified by *tid* and returns the current time. If the timer is not active, *sdl.reset* has no effect and generates no errors.

- **sdl.active** (*tid*)
  isactive, *at*

  Returns information about the status of the timer identified by *tid*. The return values are: *isactive*, a boolean indicating whether the timer is active or not, and *at*, which is the point in time at which the timer is expected to expire (if the timer is not active, then Lua's *math.huge* is returned as *at*).

System time

A LunaSDL application has a wallclock that gives the so-called **SDL system time** (or, shortly, the *system time*). The wallclock is started when the LunaSDL module is first loaded, so the system time is relative to that point in time, unless the wallclock is reset during the configuration phase.

- **sdl.now** ( )
  timestamp

  Returns the current system time, in seconds.

- **sdl.since** (*timestamp*)
  timedifference

  Returns the time elapsed from the point in time given by *timestamp*.

- **sdl.startingtime** ( )
  startingtime

  Returns the absolute timestamp corresponding to the point *0 (zero)* of the system time. The meaning of this timestamp depends on the underlying function used to retrieve time from the operating system (see *sdl.setwallclock* for more details).

Agent information

The functions described below provide information about SDL agents and help agents in locating each other.

For those functions accepting an optional *pid* parameter, this defaults to ***self***_ (i.e. to the pid of the agent invoking the function).

On error, for example if the agent searched for does not exist, all these functions return *nil* and an error message in the typical Lua way (notice that this means that they do not call Lua's *error* function, as most of other LunaSDL functions do, so the caller should check the return value).

- **sdl.pidof** (*name* [, *block*])
  pid

  Searches for the SDL agent named *name* (a string) in the SDL block identified by the pid *block*, and returns its pid. The *block* parameter is optional and defaults to ***block***_ (i.e. to the pid of the block the invoking agent is contained in).

- **sdl.nameof** ([*pid*])
  name

  Returns the name of the agent identified by *pid*.

- **sdl.blockof** ([*pid*])
  block

  Returns the pid of the block containing the agent identified by *pid*.

- **sdl.parentof** ([*pid*])
  ppid

  Returns the pid of the parent of the agent identified by *pid*.

- **sdl.kindof** ([*pid*])
  kind

  Returns the SDL *kind* of the agent identified by *pid*. The returned value (a string) is one amongst

"system", *"block"*, *"process"* or *"procedure"*.

NOTE	A procedure is not really an SDL agent, but in LunaSDL it is implemented as a special kind of agent so it is also identified by a pid and has a name.

- **sdl.stateof** ([*pid*])
  state

  Returns the current state (a string) of the agent identified by *pid*.

- **sdl.childrenof** ([*pid*])
  childrenlist

  Returns the pids of the children created by the agent identified by *pid*.

- **sdl.timersof** ([*pid*])
  timerslist

  Returns the tids of the timers owned by the agent identified by *pid*.

- **sdl.treeof** ([*pid*])

  Returns a string containing a description of the sub-tree of agents rooted at the agent identified by *pid*.

Procedures

SDL procedures are sub-parts of state machines, that can be reused in different agent scripts.

A procedure in LunaSDL is implemented as a special kind of agent, created by means of the *sdl.procedure* function, that replaces its caller from when it is called until it returns. More precisely, a procedure replaces its caller *as destination and source of signals*, i.e. all the signals addressed to the caller are redirected to the procedure, which may consume them or save them, and all the signals sent by the procedure are sent on behalf of its caller (in other words, with the caller's pid as *sender*).

When a procedure returns, all the signal it saved are automatically moved in its parent's *saved queue* and the normal addressing of signals is re-established.

Procedures can be nested, that is, a procedure may call another procedure, but agents (and procedures) can directly execute only one procedure at a time because they are replaced by it. Nested procedures all act on behalf of the original caller, whose pid they find in the ***caller_*** special variable.

State machines for procedures are defined in *procedure scripts* with the same functions used in regular agent scripts, with the exception of the *sdl.stop* function (procedure scripts shall use the *sdl.procreturn* function instead).

Another difference with regular agents is that procedures may not create timers, but they can use timers owned by the agent they act on behalf of.

- **sdl.procedure** (*atreturn, name , script, ...*)
  pid

  Executes (i.e. creates) an SDL procedure as described above. The *name*, *script* and *...* arguments are the same as in the functions that create regular agents (the only difference is that if *name=nil*, then the default name *"procedure<pid>"* is automatically assigned instead of *"agent<pid>"*).

  The *atreturn* argument defines actions to be executed in the caller agent's environment when the procedure returns. It may be a function, a string denoting a state, or *nil* if no actions need to be executed.

  When the procedure returns, if *atreturn* is a function, it is executed passing it as arguments the values returned by the procedure (if any). If *atreturn* is a string, then *sdl.nextstate(atreturn)* is automatically executed instead.

- **sdl.procreturn** (...)

  Returns from a procedure. This function is to be used in the procedure script instead of *sdl.stop*, to

terminate the procedure and possibly return values to the parent.

The arguments passed to *sdl.procreturn* (if any) are in turn passed to the *atreturn* function set by the parent when the procedure was created.

No script code should follow a call of this function.

Remote functions

Remote functions are Lua functions defined in the environment of an agent (the *exporting agent*) that can be invoked by another agent (the *invoking agent*).

When the invoking agent calls a remote function, LunaSDL switches to the exporting agent's environment, it executes the function there, and then it switches back to the invoking agent's environment returning it any value returned by the function. That is, the remote function is executed synchronously and immediately.

NOTE	This is a non-SDL construct, not to be confused with SDL *remote procedures* (see ITU-T Z.102/10.5) that have a different mechanism involving exchange of signals.

The remote function mechanism relies on the following two functions.

- **sdl.exportfunc** (*funcname*, [*func*])

  Exports the function *func* with the name *funcname* (a string), so that it can be invoked by other agents using *sdl.remfunc*.

  Calling *sdl.exportfunc* without the *func* argument revokes the function.

- **sdl.remfunc** (*pid*, *funcname*, ...)
  returnvalues

  Executes the function exported with the name *funcname* in the environment of the exporting agent identified by *pid*, and returns the function's return values (if any). The ... variable arguments, if any, are passed as arguments to the remote function.

  The invoked function must have been previously exported with *sdl.exportfunc* by the exporting agent.

The event loop

As already stated in the Overview, LunaSDL has an event loop that waits for stimuli such as the expiries of timers, or file descriptor objects that have become ready for I/O operation, and that dispatches signals sent by agents or generated by timers.

The event loop relies on a **poll function** that it uses to poll file descriptor objects for readiness and to get the timing for timers. By default, this function is LuaSocket's *select*, and the **file descriptor objects** that LunaSDL supports are LuaSocket's sockets (and compatible objects). This can be changed in the optional configuration phase by means of the *sdl.pollfuncs* function.

Agents may create and use file descriptor objects with the API provided by LuaSocket or by alternative modules, and register them in the event loop together with callbacks to be invoked whenever they are ready for I/O operations.

LunaSDL is fairly agnostic with respect to what a 'file descriptor object' is: it only expects it to be compatible with the *poll function*, and to support a couple of methods: a *settimeout* method which LunaSDL calls with a *0* timeout argument to force the object to be non-blocking, and a *__tostring* metamethod (for traces).

Multiple file descriptor objects can be used concurrently, with the only requirement that they must not be blocking, otherwise they would block the timers and the dispatching of signals (LunaSDL forces the file descriptors timeout to *0* when they are registered, and this should not be changed by the agent scripts).

Registering file descriptors

Registration and deregistration of file descriptor objects in the event loop is done with the following two functions:

- **sdl.register** (*object*, *readcallback* [, *writecallback*])

  Registers a file descriptor object in the event loop. The *object* argument must be compatible with the *poll functiont* used by LunaSDL (e.g., if the default configuration is used, then *object* must be a LuaSocket socket or a compatible object).

  If *readcallback* is not *nil*, then the event loop invokes `readcallback(object)` whenever it detects that *object* is ready for read operations. Similarly, if *writecallback* is not *nil*, then the event loop invokes `writecallback(object)` whenever it detects that *object* is ready for write operations.

  Callbacks are executed in the environment of the agent that registered the object, with the same agent set as *current* (a callback is, de facto, a transition that instead of being triggered by the arrival of an input signal, it is triggered by the detection of the readiness of a file descriptor object).

- **sdl.deregister** (*object* [, *mode*])

  Deregisters *object* from the event loop. The optional *mode* argument (a string) specifies if the object must be deregistered for read operations (*"r"*), for write operations (*"w"*), or for both (*"rw"* or *"wr"*). If not passed, *mode* defaults to *"rw"*. Deregistration of not registered objects do not cause errors.

Sockets

If the default *poll function* (i.e. LuaSocket's *socket.select*) is used, then agents may create and use sockets with the API provided by LuaSocket, registering them in the event loop with the *sdl.register* function.

Notice that the *socket. select* function supports also custom *file descriptor* objects (i.e. other than sockets), provided they have a couple of methods needed by *socket.select* to deal with them. For more details, see LuaSocket's documentation.

Event loop details

Roughly speaking, at each round the event loop does the following operations, in the order they are exposed here:

1. First, it waits for stimuli. As already stated, stimuli may be the expiry of timers, or the readiness of registered file descriptors objects for I/O operations. How long the event loop waits, it depends on the presence of signals yet to be dispatched: if there are any, it just gives a glance for stimuli (i.e. it polls with a *0* timeout) and then it goes on to the next point as soon as possible. Otherwise it waits indefinitely until the next stimulus.

2. Then, if any file descriptor object is ready for I/O, it invokes the corresponding callback. The callback execution may result in the scheduling of signals. Any such signal will be dispatched later in this same round.

3. Then, for any timer that has expired since the previous round, it schedules the corresponding signal to be delivered to the timer's owner agent. This signal will also be dispatched in this same round.

4. Finally, it dispatches all signals scheduled until now, in their order of priority. The dispatching of these signals may again result in the scheduling of new signals, but these will be dispatched in the next round (this is to avoid deadlocks).

Logs and traces

Logs

A LunaSDL application can open a **system logfile** and use the functions described here to write on it. The system logfile is also used by LunaSDL as destination for traces.

NOTE	Of course, nothing prevents an application to open other logfiles and write on them using the standard Lua libraries.

- **sdl.logopen** (*filename*)
  filehandle

  Opens the file named *filename* to be used as system logfile, and enables logs. The file is opened in write mode, using the standard Lua *io.open*. If the system logfile is already open, it is closed and re-opened. Any previously existing file with the same name is overwritten.

NOTE	If the system logfile is not open, or if it is disabled, calls of the following functions are silently ignored and raise no errors.

- **sdl.logfile** ( )
  filehandle, *filename*

  Returns the file handle and the file name of the system logfile (if it is open, otherwise it returns *nil* and an error message).

- **sdl.logson** ( )
- **sdl.logsoff** ( )

  Enable/disable logs on the system logfile, if it is open. Logs are enabled by default at the opening of the system logfile. These functions can be used to define 'logging windows'.

- **sdl.logflush** ( )

  Flushes the system logfile. LunaSDL automatically flushes it only at its closure or when the system agent stops.

- **sdl.logclose** ( )

  Flushes and closes the system logfile.

- **sdl.logf** (*formatstring*, ...)

  Formats its arguments using the standard Lua *string.format*, and writes the resulting message to the system logfile, prepending it with a timestamp and the pid of the current SDL agent.

- **sdl.printf** (*formatstring*, ...)

  Same as *sdl.logf* above, it additionally writes the message on *stdout* also (without the timestamp and the pid preamble). If the system logfile is not open or if logs are disabled, it writes on *stdout* only.

Traces

Traces are conditional logs, also written on the system logfile. Each trace is associated with a ***tag*** (a string), and it is written on the system logfile only if traces are enabled for that specific tag.

A few traces are produced by LunaSDL for troubleshooting, but application code may produce traces too.

- **sdl.traceson** (...)

  Enables traces. The function accepts an optional list of tags. If one or more tags are passed as argument, it adds them to the list of enabled tags. Otherwise it enables traces however tagged.

  By default, traces are disabled for any tag.

- **sdl.tracesoff** (...)

  Disables traces. Accepts an optional list of tags. If one or more tags are passed as argument, it adds them to the list of disabled tags. Otherwise it disables traces however tagged.

- **sdl.trace** (*tag*, *formatstring*, ...)

  Similar to *sdl.logf*, with the differences that the formatted message is written on the system logfile only if traces are enabled for the passed *tag* (a string), and that the preamble contains the tag also.

Optional configurations

The functions below allow to optionally configure some aspects of LunaSDL. If they are used, they must be called before the creation of the system agent (see the main script example).

- **sdl.envtemplate** (*env*)

  Sets the template Lua environment for environments dedicated to agents.

  The ***sdl*** global table, containing the functions described in this manual, is automatically loaded in each agent's dedicated environment, so it need not be loaded explicitly in agent scripts (unless a different name is desired for it).

  If this function is not called, the template environment is a shallow copy of the main environment (*_ENV*) at the time the *sdl.createsystem* function is called.

- **sdl.pollfuncs** (*poll*, [, *add* [, *del* [, *reset*]]])

  Sets the function to be used by the event loop to poll for stimuli, and its helper hooks.

  The passed *poll* function must have the same semantics of LuaSocket's *select*, which is used by default if *sdl.pollfuncs* is not invoked. It may support different objects than LuaSocket's sockets, but they must have a *settimeout* method accepting a *0* timeout to make them non-blocking, and a *__tostring* metamethod.

  If the optional *add* argument (a function) is passed, LunaSDL calls `add(object, mode)` whenever an object is registered in the event loop. Similarly, if the optional *del* argument (also a function) is passed, then LunaSDL calls `del(object, mode)` whenever an object is deregistered from the event loop.

  The *mode* argument passed to the *add* and *del* hooks has the same meaning and values as in the *sdl.deregister* function.

  Both the *add* and *del* hook are expected to return *true* on success, or *nil* and a string error message on failure.

  If the optional *reset* argument (also a function) is passed, LunaSDL calls `reset()` whenever it starts or re-starts the event loop.

NOTE	By means of the *add* and *del* hooks, the underlying implementation of the *poll* function can mantain internally the two sets of file descriptor objects to be polled, and avoid traversing the sets passed to it as arguments each time it is called (which it can simply ignore). The *reset* hook is expected to delete the internally mantained sets.

- **sdl.prioritylevels** ([*levels*])

  Configures the number of priority levels to be used in the LunaSDL scheduler for priority signals.

  The *levels* argument must be an integer >=*1*. If this function is not called, the number of priority levels defaults to *1* (so, by default, there is one level of priority plus the no-priority level).

- **sdl.setwallclock** ([*gettimefunc*])

  Sets the function to be used by LunaSDL to get timestamps from the underlying operating system, and resets the *SDL system time* wallclock.

  The *gettimefunc* argument is expected to be a function that, whenever called, returns a *double* representing the current time in seconds. Not passing the *gettimefunc* just resets the wallclock.

  If *sdl.setwallclock* is not called, LunaSDL uses the *socket.gettime* function provided by LuaSocket.

- **sdl.traceback** ([*s*])

  Enables the stack traceback in error messages. If *s = "off"*, it disables it. The stack traceback is disabled by default.

Examples

This section describes the examples that are contained in the `examples/` directory of the official LunaSDL release.

Some of them are accompanied with SDL diagrams, which are for illustratory purposes only.

Hello World

The first example is, of course, the traditional Hello World. Just to make things a little more interesting, the salute is given with a delay of 1 second.

The `main.lua` script loads the LunaSDL module, then creates the system agent with the `hello.lua` script. The latter implements the agent depicted in the diagram below.

The scripts for this example are in `examples/helloworld/`. The example can be run at the shell prompt with:

```
[examples/helloworld ]$ lua main.lua
```

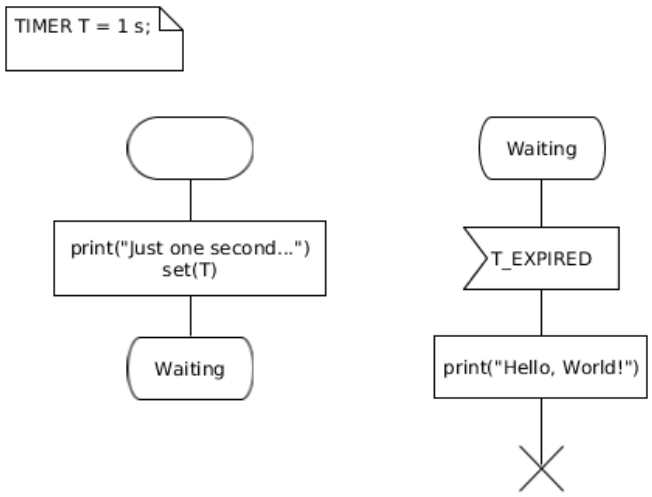

Figure 1. Hello agent

```lua
-- Main script: main.lua

sdl = require("lunasdl")

assert(sdl.createsystem("HelloSystem","hello"))
```

```lua
-- Agent script: hello.lua

local T = sdl.timer(1,"T_EXPIRED")

function Start()
   print("Just one second...")
   sdl.set(T)
   sdl.nextstate("Waiting")
end

function TExpired()
   print("Hello World!")
   sdl.stop()
end

sdl.start(Start)
sdl.transition("Waiting","T_EXPIRED",TExpired)
```

Ping Pong

In this example, the `main.lua` script creates the system agent with the `system.lua` script. The system agent then creates two processes with the same agent script, `player.lua`, and sends a `START` signal to one of them instructing it to start pinging the other, which in turns pongs in response. The system agent also sets a timer to control the duration of the ping-pong exchange, and at the timer expiry it sends a `STOP` signal to both processes and stops itself.

The scripts for this example are in `examples/pingpong/`. Run the example at the shell prompt with:

```
[examples/pingpong ]$ lua main.lua
```

(You can also pass the test duration and the ping interval as arguments, in this order).

```
-- Main script: main.lua

local sdl = require("lunasdl")

local duration = tonumber(({...})[1]) or 10 -- seconds
local interval = tonumber(({...})[2]) or 1 -- seconds

sdl.logopen("example.log")

assert(sdl.createsystem("System","system",duration,interval))
```

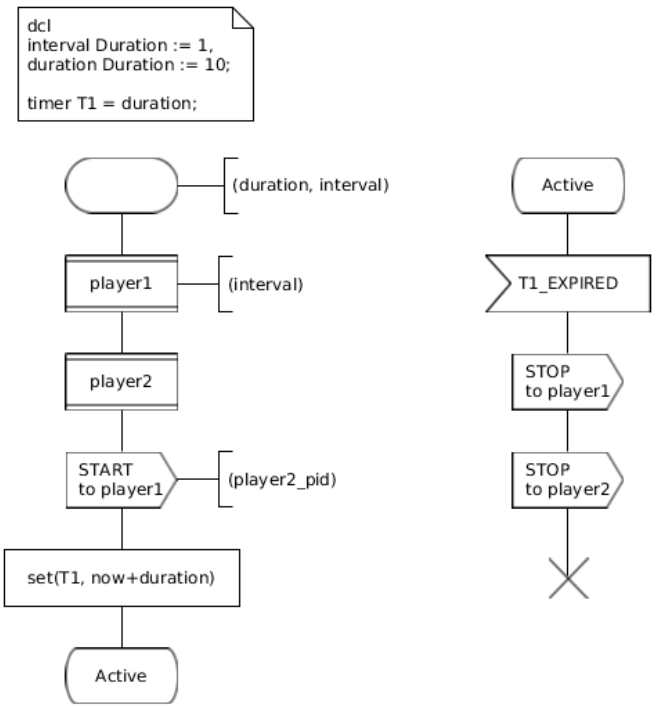

Figure 2. System agent

```
-- System agent script: system.lua

local T1 = sdl.timer(10,"T1_EXPIRED")
local player1, player2

function Start(duration, interval)
   local duration = duration or 10
   local interval = interval or 1
   sdl.printf("%s: duration=%u s, interval=%u s",name_,duration, interval)

   player1=sdl.create("player1","player",interval)
   player2=sdl.create("player2","player")

   sdl.send({ "START", player2 }, player1 )
   sdl.set(T1,sdl.now()+duration)

   sdl.nextstate("ACTIVE")
end

function Active_T1Expired()
   sdl.send({ "STOP" }, player1 )
   sdl.send({ "STOP" }, player2 )
   sdl.stop()
end

sdl.start(Start)
sdl.transition("ACTIVE","T1_EXPIRED",Active_T1Expired)
```

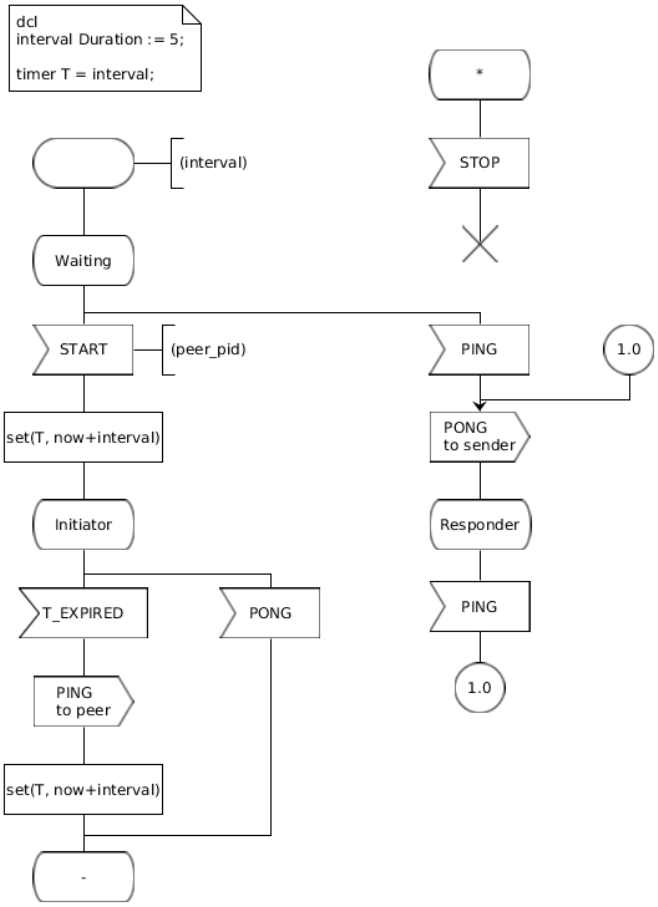

Figure 3. Player agent

```lua
-- Agent script: player.lua

local interval = 5
local T = sdl.timer(interval,"T_EXPIRED")
local peer

local function PrintSignal()
   sdl.printf("%s: Received %s from %u",name_,signame_,sender_)
end

local function Init(ping_interval)
   interval = ping_interval or interval
   sdl.nextstate("Waiting")
end

local function Waiting_Start()
   PrintSignal()
   peer = signal_[2]
   sdl.set(T, sdl.now()+interval)
   sdl.nextstate("Initiator")
end

local function Initiator_TExpired()
   sdl.send({ "PING", self_ }, peer)
   sdl.set(T, sdl.now()+interval)
end

local function Initiator_Pong()
   PrintSignal()
end

local function Waiting_Ping()
   PrintSignal()
   sdl.send({ "PONG", self_ }, signal_[2])
   sdl.nextstate("Responder")
end

local Responder_Ping = Waiting_Ping

local function Any_Stop()
   PrintSignal()
   sdl.stop()
end


sdl.start(Init)
sdl.transition("Waiting", "START", Waiting_Start)
```

```
sdl.transition("Waiting", "PING", Waiting_Ping)
sdl.transition("Initiator", "T_EXPIRED", Initiator_TExpired)
sdl.transition("Initiator", "PONG", Initiator_Pong)
sdl.transition("Responder", "PING", Responder_Ping)
sdl.transition("Any", "STOP", Any_Stop)
sdl.default("Any")
```

Ping Pong over UDP

This example reuses the `player.lua` script from the Ping Pong example.

This time the two player agents exchanging pings and pongs are created in two different systems - i.e. two different OS processes - and communicate over UDP. The system agent of each side translates messages received on the socket into signals to be forwarded to the local player agent, and viceversa.

The scripts for this example are in `examples/udppingpong/`. To run the example there are two shell scripts, `responder` and `initiator`, to be executed in this order in two different shells.

Please notice how the shell scripts define the `LUNASDL_PATH` environment variable so that LunaSDL can find the `player.lua` script in the directory containing the previous example (this is needed in this example because the script is not in the current directory).

```
-- Main script: main.lua
local sdl = require("lunasdl")

local port = tonumber(({...})[1]) or 8080
local duration = tonumber(({...})[2]) or 10 -- seconds
local interval = tonumber(({...})[3]) -- (or nil) seconds

local role = interval and "initiator" or "responder"

local ip = "127.0.0.1"
local remip = ip
local remport = port+1

if role == "initiator" then --swap UDP address
   port, remport = remport, port
   ip, remip = remip, ip
end

sdl.logopen(string.format("%s.log",role))
sdl.traceson()

assert(sdl.createsystem("system","system",ip,port,remip,remport,duration,interval))
```

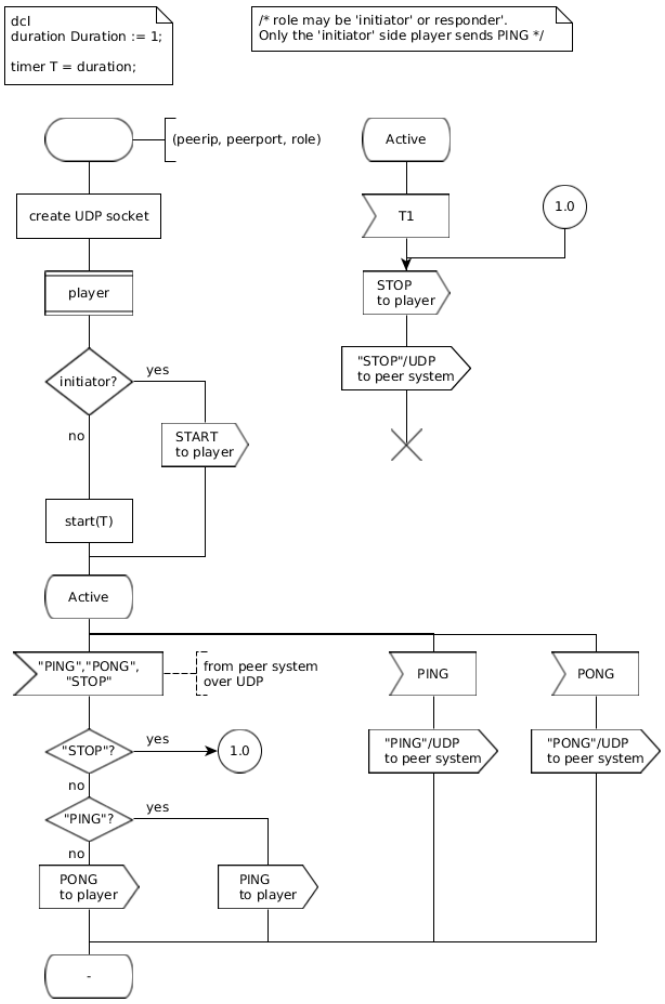

Figure 4. System agent

```lua
-- System agent script: system.lua

socket = require("socket")

local T1 = sdl.timer(10,"T1")
local player
local s, peerip, peerport, role


local function udprecv() -- socket 'read' callback
   -- get the UDP datagram and the sender's address
   local msg, fromip, fromport = s:receivefrom()
   sdl.printf("received '%s' from %s:%s",msg,fromip,fromport)

   -- check that it is an expected message
   assert(msg == "PING" or msg == "PONG" or msg == "STOP")

   -- send the corresponding signal to the local player agent
   sdl.send({ msg, self_ }, player)
   if msg == "STOP" then
      sdl.stop()
   end
end


function Start(ip, port, remip, remport, duration, ping_interval)
   peerip = remip
   peerport = remport
   role = ping_interval and "initiator" or "responder"

   sdl.printf("starting %s at %s:%s (peer system is at %s:%s)",
         role,ip,port,peerip,peerport)

   -- create a UDP socket and bind it to ip:port
   s = assert(socket.udp())
   assert(s:setsockname(ip,port))
   assert(s:setoption("reuseaddr",true))

   -- register the socket in the event loop
   sdl.register(s, udprecv)

   -- create the player agent
   player=sdl.create("player","pingpong.player", ping_interval) --

   -- send it the start signal (initiator side only)
   if role == "initiator" then
      sdl.send({ "START", self_ }, player )
```

```
   end

   -- start the overall timer
   sdl.set(T1,sdl.now()+duration)

   sdl.nextstate("Active")
end


function Active_T1Expired()
   sdl.send({ "STOP" }, player )
   s:sendto("STOP",peerip,peerport)
   sdl.stop(function () sdl.deregister(s) s:close() end)
end


function Active_Any()
   -- signal from local player, redirect signal name to peer system
   s:sendto(signame_,peerip,peerport)
end


sdl.start(Start)
sdl.transition("Active","T1",Active_T1Expired)
sdl.transition("Active","*",Active_Any)
```

```
# Responder shell script
export LUNASDL_PATH="../?.lua;;"
lua main.lua 8090 20 # port duration
```

```
# Initiator shell script
export LUNASDL_PATH="../?.lua;;"
lua main.lua 8090 10 1 # responder_port duration ping_interval
```

Web pages download

This example is a LunaSDL version of the Lua coroutines example from chapter 9 of Roberto Ierusalimschy's "Programming in Lua" (a must-read).

The `download.lua` agent script defines an agent that connects to the HTTP port of an host and downloads a file. The system agent creates four such agents - one for each desired web page - which *concurrently* download the files.

As in the PIL example, the files are not saved anywhere and the application just counts the downloaded bytes.

The scripts for this example are in `examples/webdownload/`. Run the example with... well.. you'll figure out.

(No fancy diagrams here. They would not be very interesting.)

```
-- Main script: main.lua

local sdl = require("lunasdl")

local host = "www.w3.org" -- web site where to download the pages

local pages = { -- list of desired pages
 "/TR/html401/html40.txt",
 "/TR/2002/REC-xhtml1-20020801/xhtml1.pdf",
 "/TR/REC-html32.html",
 "/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt"
}

--sdl.logopen("example.log")

assert(sdl.createsystem(nil, "system", host, pages))
```

```
-- System agent: system.lua

local ts = sdl.now()

sdl.start(function (host, pages)
   sdl.printf("%s: Creating agents", name_)
   for _,file in ipairs(pages) do
      sdl.create(nil,"download", host, file)
   end
   sdl.stop(function ()
         sdl.printf("%s: Elapsed %.1f seconds", name_, sdl.since(ts))
      end )
   end)
```

```lua
-- Agent script: download.lua

socket = require("socket")

local nread = 0 -- bytes read
local bs = 2^10 -- block size

function Callback(c) -- socket 'read' callback
   local s, status, partial = c:receive(bs)
   if status == "closed" then
      if partial then nread = nread + #partial end
      sdl.deregister(c)
      c:close()
      sdl.printf("%s: read %u bytes (finished)", name_, nread)
      return sdl.stop()
   end
   s = s or partial
   nread = nread + #s
   sdl.logf("%s: read %u bytes", name_, nread)
end

sdl.start(function (host, file)
      sdl.printf("%s: connecting to %s:80", name_, host)
      local c = assert(socket.connect(host, 80))

      sdl.register(c, Callback)

      sdl.printf("%s: retrieving '%s'", name_, file)
      c:send("GET " .. file .." HTTP/1.0\r\n\r\n")

      sdl.nextstate("_")
   end)
```

Database agent

This example uses the remote functions construct to implement a centralized database agent that serves other user agents.

The `database.lua` script defines the database agent. The `user.lua` script defines a user agent that locates the database by its agent name and uses the 'get' and 'set' functions exported by it.

The scripts for this example are in `examples/database/`.

```
-- Main script: main.lua

local sdl = require("lunasdl")

-- get the no. of entries in the database
local n_entries = tonumber(({...})[1]) or 1000

sdl.logopen("example.log")

assert(sdl.createsystem(nil,"system", n_entries))
```

```
-- System agent script: system.lua

sdl.start(function (n_entries)
   -- create the database process
   sdl.create("Database","database",n_entries)

   -- create the user process and send it a START signal
   sdl.create("User","user")
   sdl.send({ "START" }, offspring_)

   sdl.stop()
end)
```

```lua
-- Agent script: database.lua (database process)

local database = {}

function set(i,val) -- 'set' method
   sdl.printf("%s: set(%u)='%s'", name_, i, val)
   database[i]=val
   return database[i]
end

function get(i) -- 'get' method
   sdl.printf("%s: get(%u)='%s'", name_, i, database[i])
   return database[i]
end

sdl.start(function(n)
   -- populate the database
   sdl.printf("%s: populating database with %u entries", name_, n)
   for i=1,n do
      database[i] = string.format("entry no %u",i);
   end

   -- export the get/set functions for remote calls
   sdl.exportfunc("set",set)
   sdl.exportfunc("get",get)

   -- not important in this example
   sdl.nextstate("_")
end)
```

```lua
-- Agent script: user.lua (database user)

function Start()
   -- find the pid of the agent named "Database" in this block:
   local database = sdl.pidof("Database")
   assert(database,"cannot find database")
   sdl.printf("%s: Database pid is %u", name_, database)

   local ts
   ts = sdl.now()

   -- get an entry from the database
   ts = sdl.now()
   local i = 123
   local val = sdl.remfunc(database,"get",i)
   local elapsed = sdl.since(ts)
   sdl.printf("%s: entry %u is '%s' (retrieved in %.6f s)", name_, i, val, elapsed)

   -- overwrite it in the database...
   ts = sdl.now()
   sdl.remfunc(database,"set",i,string.format("hello %u %u %u",1,2,3))
   sdl.printf("%s: entry %u set in %.6f s", name_, i, sdl.since(ts))

   -- ... and then get it again
   ts = sdl.now()
   val = sdl.remfunc(database,"get",i)
   elapsed = sdl.since(ts)
   sdl.printf("%s: entry %u is '%s' (retrieved in %.6f s)", name_, i, val, elapsed)

   -- exit the LunaSDL application
   os.exit(true,true)
end

sdl.start(function (n)  sdl.nextstate("Idle") end)
sdl.transition("Idle","START",Start)
```

Priority signals

This example uses priority signals. The main script configures the number of priority levels to 3 (LunaSDL by default has only 1 level of priority, plus the *no priority* level), and then creates the system agent, which sends a few signals with different priorities to itself. Just to see their order of arrival.

The scripts for this example are in `examples/priority/`.

```
-- Main script: main.lua
local sdl = require("lunasdl")

sdl.logopen("example.log")
sdl.traceson()

-- set the number of priority levels:
sdl.prioritylevels(3)

assert(sdl.createsystem(nil,"system"))
```

```lua
-- System agent: system.lua

local function Send(n)
-- if n=nil, self-sends a signal named 'NORMAL' without priority
-- otherwise self-sends a signal named 'LEVELn', with priority n
   if not n then
      sdl.send({ string.format("NORMAL") }, self_)
   else
      sdl.send({ string.format("LEVEL%u",n) }, self_, n)
   end
end

function Start()
   -- send a few signal with different priorities to self

   -- this has no priority (i.e. lower than lowest priority):
   Send()

   -- these also have no priority because their level is higher
   -- than the configured number (=3) of priority levels:
   Send(5)
   Send(4)

   -- these have increasing priorities:
   Send(3) -- lowest (> no priority)
   Send(2) -- medium priority
   Send(1) -- highest (level 1 is always the highest)

   -- this has priority 1 because the receiver decided so
   -- by using the sdl.priorityinput function (see below)
   Send(6)

   -- finally, this also has no priority, and since signals with
   -- the same priority are dispatched first-in-first-out, it
   -- should arrive last:
   sdl.send({ "STOP" }, self_ )

   -- summarizing, the order of arrival should be the following:
   sdl.printf("expected order of arrival:")
   sdl.printf("LEVEL1, LEVEL6, LEVEL2, LEVEL3, NORMAL, LEVEL5, LEVEL4, STOP")

   sdl.nextstate("Active")
end

function Recvd() sdl.printf("received %s",signame_) end

sdl.start(Start)
```

```
sdl.transition("Active","*", Recvd)
sdl.transition("Active","STOP", function () Recvd() sdl.stop() end)

-- set the priority of 'LEVEL6' input signals to 1 (i.e. highest priority):
sdl.priorityinput("LEVEL6",1)
```

Procedure call

This example shows a procedure call and. It also uses time-triggered signals.

The system agent creates a *caller agent* with the `caller.lua` script and then sends some signals to it. The caller agent calls the procedure defined in the `procedure.lua` script. While the procedure is executing, all signals sent to the caller agent by the system agent are redirected to the procedure, which saves them. When the procedure returns, all the signals saved by it are automatically moved in the caller agent's *saved queue*. The caller restores and receives them, and then it receives other signals newly sent to it by the system agent.

The scripts for this example are in `examples/procedure/`.

(The lyrics in the code are from a catchy song by Matt Bianco and Basia.)

```
-- Main script: main.lua

local sdl = require("lunasdl")

sdl.logopen("example.log")
sdl.traceson()

assert(sdl.createsystem("System","system"))
```

```lua
-- System agent script: system.lua

local caller -- caller agent's pid
local T1 = sdl.timer(1,"T1")
local cnt = 0

local lyrics = {
   "Softly glowing, watch the river flowing",
   "It's reflections shine into my eyes",
   "I see clearer when you hold the mirror",
   "And can reach the stars up in the sky tonight",
   "(Under the moonlight, dancing in the moonlight)",
   "All around the universe, you're looking down",
   "I know you're watching over me",
   "La Luna, is it a mild case of madness?",
   "(Under the moonlight, underneath the moonlight)",
   "La Luna, you take me out of the darkness",
   "(Under the moonlight, dancing in the moonlight)"
}

function Active_T1()
   cnt = cnt+1
   local v = lyrics[cnt]
   if v then
      sdl.send({ "VERSE", v }, caller)
      sdl.set(T1)
   else
      sdl.sendat({ "STOP" }, caller, sdl.now() + 2)
      sdl.printf("%s: stopping", name_)
      sdl.stop()
   end
end

function Start()
   caller = sdl.create("Caller","caller")
   sdl.send({ "START" }, caller)
   sdl.set(T1)
   sdl.nextstate("Active")
end


sdl.start(Start)
sdl.transition("Active","T1",Active_T1)
```

```lua
-- Agent script: caller.lua

function AtReturn(...)
   sdl.printf("AtReturn(%s)",table.concat({...},","))
   sdl.restore()
end

function Active_Start()
   sdl.printf("%s: received %s", name_, signame_)
   -- call the procedure, passing it some parameters
   sdl.procedure(AtReturn, "Procedure", "procedure", "hello",1,2,3)
end

function Active_Stop()
   sdl.printf("%s: received %s", name_, signame_)
   sdl.stop()
end

function Received()
   sdl.printf("%s: received '%s' from %u",name_,signal_[2],sender_)
end

function Init()
   sdl.nextstate("Active")
end

sdl.start(Init)
sdl.transition("Active","START",Active_Start)
sdl.transition("Active","STOP",Active_Stop)
sdl.transition("Active","*", Received)
```

```lua
-- Procedure script: procedure.lua

local cnt = 0

function Waiting_Any()
   -- notice that sender_ is not self_ but caller_
   sdl.printf("%s: received '%s' from %u",name_,signal_[2],sender_)
   cnt = cnt +1
   -- save the signal for the caller:
   sdl.save()
end

function Waiting_Return()
   sdl.printf("%s: received %s from %u",name_,signame_,sender_)
   -- return from procedure, with return values:
   return sdl.procreturn("received verses",cnt)
   -- no code after sdl.procreturn()
end


function Init(...)
   sdl.printf("%s: Init(%s)",name_,table.concat({...},","))
   -- time-triggered signals are handy in procedures because procedures
   -- can not create timers...
   sdl.sendat({ "RETURN" }, self_, sdl.now() + 5)
   sdl.nextstate("Waiting")
end


sdl.start(Init)
sdl.transition("Waiting","RETURN",Waiting_Return)
sdl.transition("Waiting","*",Waiting_Any)
```

Special variables

In LunaSDL applications, identifiers composed of one or more lower-case letters and ending with an underscore (e.g., *self_*) are **reserved for special variables** managed by the LunaSDL module, and should not be used in agents' scripts code.

The special (global) variables listed below are always set, and are properly updated every time (and before) a transition is executed.

Variable	Description
self_	the pid of the agent itself.
name_	the name of the agent.
block_	the pid of the SDL block the agent is contained in.
parent_	the pid of the parent agent.
offspring_	the pid of the last child agent created by this one (or *nil* if none).
state_	the current state name (or *nil* if the agent is stopping).
signal_	the current input signal.
signame_	the name of the input signal (same as *signal_[1]*)
sender_	the sender of the input signal (pid, or tid if the signal is timer-generated).
caller_	the pid of the original caller of the procedure (*nil*, if the current agent is not a procedure).
istimer_	a boolean indicating if the input signal was generated by a timer.
sendtime_	a timestamp indicating when the signal was sent.
recvtime_	a timestamp indicating when the signal was received.
exptime_	a timestamp indicating when the signal becomes stale.

All the above variables are to be considered **read-only** (care should be taken not to change their values from agent script code).

NOTE

The ***sendtime_*** timestamp refers actually to the point in time the signal was scheduled for dispatching. If the signal was sent with *sdl.send*, this corresponds to when the function was called; for timer-generated signals, it is the point in time the timer expiry was detected; for time-triggered signals it is the point in time the signal was actually delivered.

Summary of LunaSDL functions

The tables below summarize the functions provided by LunaSDL.

With some exceptions pointed out in the manual sections, the behavior of all the functions when an error occurs is to call Lua's *error* function with a string message.

Table 1. Create functions

Function	Return values	Description
sdl.createsystem (*name* , *script*, ...)	*true*, *returnvalues*	Creates the *system agent*.
sdl.createblock (*name* , *script*, ...)	*pid*	Creates an agent of the kind *block*.
sdl.create (*name* , *script*, ...)	*pid*	Creates an agent of the kind *process*.
sdl.systemreturn (...)	-	Sets the system agent return values.

Table 2. State transition functions

Function	Return values	Description
sdl.start (*func*)	-	Sets the agent's *start transition*.
sdl.transition (*state*, *signame*, *func*)	-	Sets a transition.
sdl.default (*state*)	-	Sets the agent's default state.
sdl.nextstate (*state*)	-	Changes the agent's state.
sdl.stop ([*atstopfunc*])	-	Stops the agent.
sdl.kill ([*pid*])	-	Kills an agent and all its descendants.

Table 3. Signals functions

Function	Return values	Description
sdl.send (*signal*, *dstpid* [, *priority*])	*now*	Sends a signal.
sdl.priorityinput (*signame*, [*priority*])	-	Sets the priority for an input signal.
sdl.sendat (*signal*, *dstpid*, *at* [, *maxdelay*])	-	Sends a time-triggered signal.
sdl.save ( )	-	Saves the current input signal.
sdl.restore ( )	-	Re-schedules the saved signals.

Table 4. Timers functions

Function	Return values	Description
sdl.timer (*duration*, *signame*)	*tid*	Creates a timer.
sdl.modify (*tid*, *duration* [, *signame*])	-	Modifies a timer.
sdl.set (*tid* [, *at*])	*now*	Sets (starts) a timer.
sdl.reset (*tid*)	*now*	Resets (stops) a timer.
sdl.active (*tid*)	*isactive*, *at*	Returns the status of a timer.

Table 5. System time functions

Function	Return values	Description
sdl.now ( )	*timestamp*	Returns the current system time.
sdl.since (*timestamp*)	*timedifference*	Returns the time elapsed since a point in time in the past.
sdl.startingtime ( )	*startingtime*	Returns an absolute timestamp corresponding to system time = *0*.

Table 6. Agent information functions

Function	Return values	Description
sdl.pidof (*name* [, *block*])	*pid*	Searches for an agent by its name.
sdl.nameof ([*pid*])	*name*	Returns the name of an agent.
sdl.blockof ([*pid*])	*block*	Returns the pid of an agent's block.
sdl.parentof ([*pid*])	*ppid*	Returns the pid of an agent's parent.
sdl.kindof ([*pid*])	*kind*	Returns the SDL *kind* of an agent.
sdl.stateof ([*pid*])	*state*	Returns the current state of an agent.
sdl.childrenof ([*pid*])	*childrenlist*	Returns the pids of an agent's children.
sdl.timersof ([*pid*])	*timerslist*	Returns the tids of an agent's timers.
sdl.treeof ([*pid*])	*treestring*	Returns a description of the sub-tree of an agent.

Table 7. SDL procedure functions

Function	Return values	Description
sdl.procedure (*atreturn*, *name* , *script*, ...)	*pid*	Executes an SDL procedure.

Function	Return values	Description
sdl.procreturn (...)	-	Returns from a procedure.

Table 8. Remote functions

Function	Return values	Description
sdl.exportfunc (*funcname*, [*func*])	-	Exports or revokes a function.
sdl.remfunc (*pid*, *funcname*, ...)	*returnvalues*	Executes a remote function.

Table 9. Socket registration

Function	Return values	Description
sdl.register (*object*, *readcallback* [, *writecallback*])	-	Registers a file descriptor in the event loop.
sdl.deregister (*object* [, *mode*])	-	Deregisters a file descriptor from the event loop.

Table 10. Log functions

Function	Return values	Description
sdl.logopen (*filename*)	*filehandle*	Opens the system logfile.
sdl.logfile ( )	*filehandle*, *filename*	Returns the system logfile handle and name.
sdl.logson ( )	-	Enables logs in the system logfile.
sdl.logsoff ( )	-	Disables logs in the system logfile.
sdl.logflush ( )	-	Flushes the system logfile.
sdl.logclose ( )	-	Flushes and closes the system logfile.
sdl.logf (*formatstring*, ...)	-	Writes on the system logfile.
sdl.printf (*formatstring*, ...)	-	Writes on the system logfile and on *stdout*.
sdl.traceson (...)	-	Enables traces.
sdl.tracesoff (...)	-	Disables traces.
sdl.trace (*tag*, *formatstring*, ...)	-	Writes a trace on the system logfile.

Table 11. Optional configuration functions

Function	Return values	Description
sdl.envtemplate (*env*)	-	Sets the template environment.
sdl.pollfuncs (*poll* [, *add* [, *del* [, *reset*]]])	-	Sets the poll function and its helper hooks.
sdl.prioritylevels ([*levels*])	-	Configures the number of priority levels.
sdl.setwallclock ([*gettimefunc*])	-	Sets the *gettime* function and resets the wallclock.
sdl.traceback ([*s*])	-	Enables the stack traceback in error messages.