Aloha from Lua

Table of Contents

Introduction

Lua is a C-based embedded[1] scripting language that has found popularity in games and other applications where providing the ability for end-users to customize applications easily and rapidly is highly desirable. Lua is not only written in C, it is intended to be highly *inter-operable* with C – you can call Lua programs from C, and vice-versa. But Lua can also be used stand-alone.

An introduction to Lua on the Propeller 2 is given in the document **Lua on the Propeller 2 with Catalina**. That document also contains a description of Catalina's multi-processing extensions to Lua, which enable Lua to take advantage of multiple cogs on multiple threads.

This document is about a *different* extension to Lua that Catalina offers - **eLua** offers a simple way to execute two instances of Lua on the same Propeller using a simple client/server architecture, which provides another way for Lua to take advantage of the Propeller's unique capabilities.

eLua allows Lua to get around both the speed and space limitations that could otherwise make Lua impractical on a microcontroller. Lua programs can potentially have both *speed* and *space*, as well as the *multi-processor* capabilities offered by the Propeller.

eLua programs can also be easily extended to multiple propellers, using a very simple serial protocol called **ALOHA**. First, this document will describe **eLua** by itself, and then **eLua** with **ALOHA**.

eLua

The main purpose of **eLua** is to enable multiple Lua instances to execute in parallel. Typically, one Lua instance will execute from Hub RAM at full propeller speed but with limited program and data space, while the other Lua instance will execute from XMM RAM, at slower speeds but with megabytes of program and data space.

The interaction between the two Lua instances is via a very simple client/server architecture, where the server provides one or more Lua functions that the client can call when it needs something done that would take too much code or data space for it to do itself. The server can also run a "background" task when it is not servicing function calls from the client, allowing a simple and natural way to implement multi-processing applications.

[1] The term "embedded" here means embedded within other programs, not embedded in hardware.

Building eLua

There are several variants of essentially the same **eLua** client/server program in the folder *demos/elua* - two main ones, plus a few others that may be useful in particular circumstances.

Here is a brief description of the two main **eLua** variants:

elua	a CMM Lua client with Lua multi-processing support enabled, and an XMM LARGE Lua server with an 8K cache, both *with* a Lua parser.
eluax	a CMM Lua client with Lua multi-processing support enabled, and an XMM LARGE Lua server with a 64K cache, both *without* a Lua parser.

The other **eLua** variants omit the multi-processor support, which saves Hub RAM when these capabilities are not required:

eluafx	an NMM Lua client and an XMM LARGE Lua server with an 8K cache, both without a Lua parser or multi-processing support. Because the client is a Native program, it will execute faster than the other eLua variants - but there is less Hub RAM available, so only smaller Lua clients (or servers, depending on how the Hub RAM is allocated) can be executed.
eluas	a CMM Lua client and an XMM LARGE Lua server with an 8K cache, both with a Lua parser, but without multi-processing support.
eluasx	a CMM Lua client and an XMM LARGE Lua server with a 64K cache, both without a Lua parser or multi-processing support.

Compile them all using Catalina Catapult by setting **CATALINA_DEFINE** to the appropriate propeller platform and executing the **build_all** script, or compile them individually by using the following Catapult commands (this example is for the P2_EDGE):

```
set CATALINA_DEFINE=P2_EDGE
catapult elua.c
catapult eluax.c
catapult eluas.c
catapult eluafx.c
catapult eluasx.c
```

The variants differ only in the attributes of the various Catapult pragmas, which tell Catalina what memory model to use for the primary (server) and secondary (client) programs, their stack sizes, addresses, cache size, etc.

All the eLua variants should build "as is" for a P2 Edge with on-board PSRAM (i.e. the P2-EC32MB), or a P2 Evaluation board with the HyperRAM add-on board. With minor modification - typically, to the address to use for the client program - they can also be built for other Propeller 2 platforms with supported PSRAM. As usual with Catapult programs, the programs will tell you the address to use when they are compiled and executed if this needs to be changed.

All the **eLua** variants load the client and server Lua files specified on the command line - the first argument specifies the client file, and the second specifies the server file. The variants that include a Lua parser (such as **elua**) can accept either text or compiled binary Lua files (e.g. *client.lua* or *client.lux*). The variants that do *not* include a Lua parser (such as **eluax**) can accept only compiled binary Lua files (e.g. *client.lux*). If no files are specified, the defaults are *client.lua* and *server.lua* for those variants that include a Lua parser, and *client.lux* and *server.lux* for those that do not.

All **eLua** programs use the Catalina Registry for their client/server interaction - this enables the server to offer multiple services, and also adds lock protection so that the services can be used safely in multi-processing applications.

The Catalina registry supports only a limited number of service parameter profiles - basically, those that are required to implement Catalina plugins.

However, the basic "short" service profile can also be used to pass a pointer to the shared data structure as the parameter. This allows for arbitrary data to be exchanged between client and server. In **eLua**, the shared data is primarily used to store and share the names of the client and server files, and also to synchronize the startup between the client and the server (so that the client does not try to call services provided by the server before the server is ready).

Also, specifically for Lua a "serial" service profile has been added that can be used to accept and return any simple Lua data type. They are passed in a "serialized" format. A binary serialize/deserialize library (called **binser**) is provided for this purpose. Any Lua function can be implemented using the serial service type, because it can accept and return one or more of any of the basic Lua data types - including Lua functions. This means a client can not only exchange arbitrary data with the server - it can also exchange functions with the server for remote execution (the simple demo provided has an example of doing this). The **binser** module is described in more detail in the **Technical Notes** section of this document.

Note that the **eLua** programs DO NOT generally need to be recompiled just to execute different Lua programs. The main reasons **eLua** would need to be recompiled would be to adjust the allocation of Hub RAM between the Lua client and the Lua server, to alter plugins used, or to alter the default modules loaded by Lua. See the **Technical Notes** section of this document for more detail on Hub RAM allocation.

Demonstrating eLua

To demonstrate **eLua**, after building the binaries copy the following files from *demos/elua* to an SD card containing Catalyst:

elua.bin	the eLua program
eluax.bin	the eLuax program

Then add the files from the *demos/elua/example* folder:

binser.lua	the **binser** module
common.lua	common definitions for the clients and servers
client.lua	an example eLua client
server.lua	an example eLua server
remote.lua	an ALOHA proxy client/server (more on this later)
serverbg.lua	an example eLua server with a background task
rebuild	a Catalyst script to re-compile all the Lua files

plus compiled binary (*.lux*) versions of the Lua files.

These files implement an example **eLua** program. Since the files have the names that will be loaded by default, to execute the example simply execute any of the eLua variants without specifying any parameters. For example:

```
elua
```

or

```
eluax
```

This example implements five different services, intended to illustrate various **eLua** features:

add	accepts two numbers and returns their sum. Demonstrates that a service can accept multiple arguments.
modrem	accepts two numbers and returns their modulus and remainder. Demonstrate that a service can return multiple results.
invert	accepts a Lua table and returns a Lua table that "inverts" the value of each table element. Demonstrates that a service can accept and return a Lua table.
invoke	accepts a Lua table that contains a number and a Lua function, invokes the function on the number, and returns the result. Demonstrates that a service can accept any simple Lua data type, including Lua functions.
quit	causes the server to terminate. Provides a mechanism for orderly program termination.

The *common.lua* file contains the definition of all the services required by the *client.lua* program, and *proxy* functions that can be used to call them. The *server.lua* program is slightly more complex, but essentially just adds a *wrapper* function around each service that allows it to be called by the Lua dispatcher. The client doesn't need to know any of these details - it simply calls the proxy functions as if they were local functions and is unaware that they may actually be implemented by the server and not the client.

Note that if you change any of the Lua text files, you should also recompile the corresponding binary versions. For example:

```
luac -o client.lux client.lua
```

A catalyst script to recompile all the Lua binary versions is provided called **rebuild**. To execute it use the following Catalyst command:

```
exec rebuild
```

Instead of discussing this example in detail, it is better to start with a simpler demo. The simple demo given later in this document functions perfectly well when executed as an **eLua** example with or without ALOHA. So it is possible to skip straight from here to the section titled **A simple demo** and then come back later to find out more about **eLua**'s multi-processing support and ALOHA.

Multi-Processing with eLua

This section describes using eLua to execute programs that are not specifically **eLua** programs. They are the example programs described in the document **Lua on the Propeller 2 with Catalina**, and should already be on the Catalyst SD card (if not, copy them from *demos/catalyst/lua-5.4.4/test*). However, they can be executed as client programs by **eLua** and can also be executed in conjunction with an **eLua** server that offers additional services. For example:

```
elua ex6.lua
```

or

```
luac -o ex6.lux ex6.lua
eluax ex6.lux
```

These programs do not themselves use the client/server capabilities of eLua, but because the server code will be loaded and executed even if it is not used, some of the demos need minor tweaks due to the reduced Hub RAM and cogs available to **eLua** clients (and they must all be executed as **eLua** clients, because **eLua** servers do not support multi-processing capabilities). Here are the tweaks required:

ex1.lua	must be executed by **elua** since it requires the Lua parser.
ex8.lua	can execute at most 5 Lua threads (not 10).
ex9.lua	can use at most 2 factories (not 4), and can recycle at most 2 workers (not 4).
ex12.lua	can use at most 2 factories (not 4) and 2 workers (not 4).

Also, note that these demos usually end with a **Press ENTER to terminate** message - but this will only terminate the client and not the server because these programs are not aware the server needs to be terminated. Instead, reset the Propeller to return to the Catalyst prompt.

To verify that the server is actually functioning (and can be used for other tasks) an alternate server that flashes a LED periodically is provided. This is in the file *serverbg.lua* (or its compiled equivalent *serverbg.lux*).

To see it in action, load any of the examples (except *ex10.lua*, which expects to use the LEDs itself) but specify *serverbg.lua* (or *serverbg.lux)* instead of loading the default. For example:

```
elua ex1.lua serverbg.lua
```

or

```
eluax ex6.lux serverbg.lux
```

The serverbg server can also be used in place of the normal server when executing the **eLua** example program, since it implements the same services in addition to executing a background task. For example:

```
elua client.lua serverbg.lua
```

ALOHA

ALOHA extends **eLua** to allow programs to execute clients and servers on different propellers.

The best way to get started with ALOHA is with an example. In this section, we will walk through a simple - but fully functional - **eLua**/ALOHA demo[2]. This demo can be executed as an **eLua** program on one propeller, or as an ALOHA program on two propellers. If it is only going to be executed on a single propeller, the sections below titled **Preparing multiple propellers** and **Building eLua with ALOHA** can be skipped.

Preparing multiple Propellers

ALOHA is intended to be used on multiple propellers. The terms 'master' and 'slave' are used for these rather than 'client' and 'server' because when running **eLua**, each propeller runs both a local client and a local server, so we need to distinguish between the client and server running on the master and the clients and servers running on each of the slaves.

Specifically to support this type of master/slave multi-propeller programming, two new platform configuration files have been added to the *target/p2* folder, and the *target/p2/platforms.inc* file - they are called *P2_MASTER.inc* and *P2_SLAVE.inc*.

The propellers must be connected using two pins for serial communications - the default is to have a dedicated serial connection between the master and each slave, although "multi-dropping" - where a single serial connection is used to connect the master with all the slaves is also possible, provided each slave implements a different set of services.

The 2 or 8 port serial plugin can be used, and the **SIMPLE** HMI option can be used by each propeller (the **TTY** HMI option can be used if the 8 port serial plugin is used, but not if the 2 port serial plugin is used). For this demo we will use the 2 port serial plugin on both the master and slave propellers, with the propellers connected as shown below:

```
P2_MASTER    P2_SLAVE
   rx 0 <——---------> 0 tx
   tx 1 <------------> 1 rx
```

So the file *target/p2/P2_MASTER.inc* should specify:

```
' 2 Port Serial constants
' =======================
_RX1_PIN    = 0
_TX1_PIN    = 1
```

and the file *target/p2/P2_SLAVE.inc* should specify:

```
' 2 Port Serial constants
' =======================
_RX1_PIN    = 1
_TX1_PIN    = 0
```

[2] A more complete version of the demo, with a few more "bells and whistles" is included in **Appendix A**.

The *P2_MASTER.inc* and *P2_SLAVE.inc* files included are copies of *P2_EDGE.inc*, modified as above. If you have a different propeller platform, modify them accordingly.

A picture is probably worth a thousand words at this point, so here is one:

```
+----- Master -----+
|   +----------+   |
|   |          o----- port n -----------+
|   |  client  |   |   ...              |
|   |          o----- port 0 --+        |
|   +----------+   |           |        |
|        |         |           |        |
|        |         |           |        |
|   local calls    |           |        |
|        |         |           |   ...  |
|        V         |           |        |
|   +----------+   |           |        |
|   |  server  |   |           |        |
|   +----------+   |           |        |
+---- Propeller ---+           |        |
                            remote      |
                        procedure calls |
                               |        |
+----- Slave 0 ----+           |        |
|   +----------+   |           |        |
|   |  client  |   |           |        |
|   +----------+   |           |        |
|        |         |           |   ...  |
|        |         |           |        |
|   local calls    |           |        |
|        |         |           |        |
|        V         |           |        |
|   +----------+   |           |        |
|   |  server  o<---- port 0 --+        |
|   +----------+   |                    |
+---- Propeller ---+                    |
                                        |
         .                           remote
         .                       procedure calls
         .                              |
                                        |
+----- Slave n ----+                    |
|   +----------+   |                    |
|   |  client  |   |                    |
|   +----------+   |                    |
|        |         |                    |
|        |         |                    |
|    local calls   |                    |
|        |         |                    |
|        V         |                    |
|   +----------+   |                    |
|   |  server  o<---- port 0 -----------+
|   +----------+   |
+---- Propeller ---+
```

Because the demo program has to be built to execute on two or more different propeller platforms (or two or more of the same platform but perhaps with different options and/or configurations) the easiest way to build them is using Catapult, after setting **CATALINA_DEFINE** to the appropriate platform. A **build_all** script to do this is provided (this is described further below).

Building eLua with ALOHA

To execute the demo on one Propeller, we do not need to do anything further - we can just execute it with **elua** or **eluax**. Skip to the next section.

To execute the demo on multiple propellers, we must build **eLua** for both the master and the slave platforms, and build both the master and slave versions to include ALOHA.

As previously described, building **elua** (and **eluax**) can be most easily done using Catapult, after defining **CATALINA_DEFINE** to specify the appropriate platform, as follows:

```
set CATALINA_DEFINE=P2_SLAVE
catapult elua.c
catapult eluax.c
```

or

```
set CATALINA_DEFINE=P2_MASTER
catapult elua.c
catapult eluax.c
```

The **build_all** script in the folder *demos/elua/aloha* does exactly this, but it also renames the resulting P2_MASTER binaries as *master.bin* and *masterx.bin*, and the P2_SLAVE binaries as *slave.bin* and *slavex.bin* - so we will end up the following four binaries:

master.bin	**eLua** compiled for P2_MASTER
masterx.bin	**eluax** compiled for P2_MASTER
slave.bin	**elua** compiled for P2_SLAVE
slavex.bin	**eluax** compiled for P2_SLAVE

Then **master** (or **masterx**) must be executed on the P2_MASTER propeller, and **slave** (or **slavex**) must be executed on the P2_SLAVE propeller. The **master** and **slave** names are arbitrary - in fact all the binaries are simply **eLua**, but built for a specific platform, and including a custom Lua dispatcher and the ALOHA protocol (both described later in this document).

The *elua.c* and *eluax.c* programs are configured to use the 2 port serial plugins by default, but the 8 port serial plugin can be used if more than 3 propellers are to be linked in a star configuration (with one master connected to up to eight slaves - see the diagram above).

To use the 8 port serial plugin, simply specify **-lserial8** in place of **-lserial2** before compiling *elua.c* and *eluax.c* (i.e. modify the Catapult pragmas accordingly). It is also possible to use the 2 port plugin on the servers and the 8 port plugin on the client.

A simple demo

Loading the simple demo

Copy the appropriate **eLua** binaries to the appropriate Catalyst SD cards (for two propellers, one is needed for the P2_MASTER propeller and one for each P2_SLAVE propeller).

To each SD card also copy the contents of the *demos/elua/simple* folder, which will include:

binser.lua	the **binser** module
common.lua	common definitions for the clients and servers
client.lua	an example eLua client
server.lua	an example eLua server
remote.lua	an ALOHA proxy client/server
rebuild	a Catalyst script to re-compile all the Lua files

plus compiled (i.e. *.lux*) versions of the Lua files. Note that these files have the same name as the ones in the **eLua** example - those files will be overwritten if they already are on the SD card.

Note that if you change any of the Lua text programs, you should also recompile the corresponding binary versions. For example:

```
luac -o client.lux client.lua
```

A catalyst script to recompile all the Lua binary versions is provided called **rebuild**. To execute it use the following Catalyst command:

```
exec rebuild
```

Executing the simple demo

To execute the demo on a single propeller, just execute **elua** (or **master** or **slave**) with no parameters.

They will all execute *client.lua* and *server.lua* as the **eLua** client and server, and the client calls functions in the local server - i.e. in the same Propeller. This will not use ALOHA.

To execute the demo on *multiple* propellers, do the following:

First[3], on the P2_SLAVE, execute **slave** but specify **remote.lua** as the *client*:

```
slave remote.lua server.lua
```

Then, on the P2_MASTER, execute **master** but specify **remote.lua** as the *server*:

```
master client.lua remote.lua
```

[3] In this simple version the commands *must* be executed in this order - the slave has to be ready *before* the master starts or it will time out. This is not true in the more complete version given in Appendix B.

The client on the master propeller will use ALOHA to call the server on the slave propeller.

The simple demo files

This section contains a walk-through of all the significant lines in all the files in the *demos/elua/simple* folder. Each file is only a few lines long - to see them all on a single page, see **Putting it all together**, below.

client.lua

The file *client.lua* contains only two lines of significance.

The first loads definitions required by all clients and all servers:

```
dofile 'common.lux'
```

This line simply executes *common.lux*, which is the compiled version of *common.lua* (do not jump ahead just yet to read the section on *common.lua* - read the sections in the order they are included in this document - they are given in this order for a reason).

This line could actually specify *common.lua* - however, it is generally recommended to use the *compiled* version of the common module (i.,e. *common.lux*) rather than the *text* version (i.e. *common.lua*) because the former will work even if *client.lua* is subsequently compiled (e.g. to *client.lux*) and then executed by a version of **eLua** that does not include the Lua parser, whereas the latter will not (it will report an error). However, using the text version is useful while developing programs, as it does not need to be recompiled after each amendment.

The only other significant line is the one that shows how the client calls a server function. In the following line, a perfectly ordinary Lua function called **invoke** is being called, and the result of the call is then printed:

```
print(invoke(function(x) return x*x end, 2.5))
```

The **invoke** function may be familiar. It is essentially the same as the one in the non-ALOHA **eLua** demo. It accepts two parameters:

- a function that accepts a value; and
- a value to be passed to that function.

If this line is a little cryptic, it is very likely because in Lua, functions are *first class types*, and can be specified anywhere any other data type can be specified. Most languages do not support this, and treat functions as second class citizens that must be specially defined - generally *before* they can be used anywhere.

Lua can do that too, of course. The line above could equally well be written as shown below, which may make it clearer that a function **f** that takes a parameter called **x** and returns a result (**x*x** in this case) is being passed to **invoke**, along with a value for **x**:

```
function f(x)
  return x*x
end
x = 2.5
result = invoke(f, x)
```

Neither the names **f** nor **x** are significant here. Consider the line below - neither **f** nor **x** appear, but this line is also perfectly acceptable. Can you predict what it will do if it is executed as part of *client.lua*?

```
invoke(function(str) print(str) end, "ALOHA"))
```

We can include this line in *client.lua* to find out.

server.lua

The file **server.lua** is a little more complex. But it also contains only a few lines of significance.

Again, the first one loads definitions required by all clients and all servers:

```
dofile 'common.lux'
```

The next significant line defines serial port usage - it defines a table of functions that will be called via the specified serial port.

One such entry is required for each ALOHA port that might call this server, called **port_0_index** .. **port_7_index**. An empty table indicates the port is to be *monitored* for inward function calls, but no remote calls are made *out* of this port. In our example program, we only want to monitor port 0 for incoming remote calls, so we need only one line:

```
port_0_index = { }
```

The next significant lines are those that define the **invoke** function, which is the function that clients will call remotely:

```
function invoke(serial)
   f, x = bs.deserializeN(serial, 2);
   output = f(x);
   return bs.serialize(output)
end
```

This is not the normal Lua function definition we might have expected for **invoke**. First, it accepts only one parameter called **serial** (rather than the two parameters **f** and **x** we might have expected). It calls **deserializeN** on that parameter to get the actual parameters **f** and **x**, then it invokes **f** on **x** (which is what we would have expected), and then it calls **serialize** on the output before it returns it.

This (apart from the line which calls the actual function) is how *all* services must be written. Although the service appears to accept just one serialized argument and return one serialized value, those serialized values can actually encode multiple *actual* values.

When the function **invoke** is called by the client, the client actually calls the *proxy* function (defined in **common.lua**) which passes a *serialized* version of all its parameters (which in this case are the 2 parameters **f** and **x**). The function **deserializeN** *deserializes* the specified number of parameters from its input (again, in this case the 2 parameters **f** and **x**). The next line simply invokes the function **f** on parameter **x** and generates output, and the last line calls **serialize** to turn the result into a form that can be passed back to the caller.

This will become clearer when we look at the proxy function for **invoke** defined in **common.lua**.

The important point here is that the function arguments and function return values are always passed between **eLua** clients and servers - whether those clients are on the same propeller or on different propellers - in a *serialized* binary format. As we have seen, these arguments can be any simple Lua data type, including functions and tables. They are *serialized* before transmission, and must be *unserialized* on reception. This is accomplished using functions provided by the package **binser**, which is a general purpose Lua *serializer* and *deserializer*.

A detailed discussion of **binser** is beyond the scope of this document, but a very quick summary of all we need to know is given in the section titled **The binser module** - and the example above illustrates this is quite easy.

The last significant lines are those that define what the server should do when it is not servicing remote calls. This is a special function called **background**. In this case, we don't need the server to do anything except respond to remote calls to **invoke**, so it is just a null procedure.

```
function background()
end
```

A more detailed discussion of background tasks is given in the section titled **Background tasks**.

remote.lua

This file is only relevant when using ALOHA. In this example program, the file **remote.lua** is almost completely trivial, and contains only a few lines of significance.

Again, the first one loads definitions required by all clients and all servers:

```
dofile 'common.lux'
```

And the only other significant lines are those that define the port that will be monitored for service calls, and the calls that are expected on that port. In this case, we expect only one call to be made, and they must be made via port 0, to our **invoke** service:

```
port_0_index = {
  [INVOKE_SVC] = "invoke"
}
```

Contrast *this* instance of **port_0_index** with the previous instance, described in **server.lua**. That instance was for the *slave* server, which *accepts* calls on port 0 but doesn't make any, and this instance is for the *master* server, which actually makes such calls.

common.lua

The **common.lua** file is where most of the 'magic' happens.

First, of all, there is some necessary housekeeping:

```
svc = require 'service'
bs  = dofile 'binser.lux'
```

The **service** module is required to allow **eLua** programs to interact with the Catalina Registry. Very briefly, the Registry provides the necessary mechanisms to allow communication between two Lua programs, whether they are running on different cogs on the *same* propeller, or on *different* propellers.

The **binser** module is used by all **eLua** clients and servers to serialize and deserialize function arguments and return values. See the section titled **The binser module** for more details.

The next significant lines are where we define a Catalina service id for each service we intend to provide (in this case there is only one - for the **invoke** service).

```
INVOKE_SVC = 81
```

Why **81**? Well, all service requests go via the Registry, even if the service is to be provided by another propeller, and Catalina supports service ids in the range 0 .. 255. But services ids 1 .. 80 are reserved for Catalina's own use. So eLua service ids must start from 81.

The next significant lines define an index of services the server will provide, as a Lua table called **service_index**. In this case, we need only one entry, again for the **invoke** service:

```
service_index = {
  [INVOKE_SVC] = "invoke"
}
```

Finally, we define a *proxy* service for each of the functions we want the client to be able to call. Again, in this case we need only one, for the **invoke** service:

```
function invoke(f, x)
  return bs.deserializeN(svc.serial(INVOKE_SVC, bs.serialize(f, x), 500), 1)
end
```

Not the use of **binser** again here. This instance of **invoke** is a proxy for the actual **invoke** function in **server.lua** (which is where the real function actually executes). This proxy function is what makes the whole thing work - it allows clients to call the **invoke** function exactly as they normally would, without having to know where that function is executed. It may be executed on the local master server, or it may be executed on a remote slave server.
The parameter 500 (to **svc.serial**) specifies how large a buffer may be required to be allocated internally to accommodate the serialize/deserialize process. An error will be reported if the buffer is not large enough.

Putting it all together

Every significant line in the simple demo has been explained in the previous sections. But it may be easier to understand when it is seen all together on one page - so here it is again, in full:

common.lua:

```
svc = require 'service'
bs  = dofile 'binser.lux'

INVOKE_SVC = 81

service_index = {
  [INVOKE_SVC] = "invoke"
}

function invoke(f, x)
  out = bs.deserializeN(svc.serial(INVOKE_SVC, bs.serialize(f, x), 500), 1)
  return out
end
```

client.lua:

```
dofile 'common.lux'

invoke(function(str) print(str) end, "ALOHA")
print(invoke(function(x) return x*x end, 2.5))
```

server.lua:

```
dofile 'common.lux'

port_0_index = { }

function invoke(serial)
   f, x = bs.deserializeN(serial, 2)
   output = f(x)
   return bs.serialize(output)
end

function background()
end
```

remote.lua:

```
dofile 'common.lux'

port_0_index = {
  [INVOKE_SVC] = "invoke"
}

function background()
end
```

Technical Notes

File name conventions

An **eLua** program with one client and one server typically consists of three files. They can be given any names, but the following convention is recommended:

common.lua	definitions common to all the clients and servers in the eLua program
client.lua	the master client Lua program.
server.lua	the slave server Lua program.

eLua programs that use **ALOHA** can span multiple propellers, and at least one more file is typically required:

remote.lua	the master server Lua program, which often may also conveniently be used as the slave client program.

If there is more than one slave (i.e. a multi-propeller **eLua** program with more than two propellers), then adding suffixes **_0** .. **_7** to the server and remote file names is recommended.

Following this convention means that the common file is the only one that needs to know the details of the service ids required to implement the client/server communications, and also provides a place to implement the proxy services that allow the client to be unaware that the services are not implemented locally. It also provides a convenient place to load the common modules (e.g. **binser**) and perform any necessary client or server configuration.

The service ids (a unique one must be allocated to each service) can be any service id other than those reserved for Catalina's own use (see *target/p2/constants/inc*) - i.e. from (**SVC_RESERVED** + 1) to 255 - ii.e. 81 to 255.

Lua initialization modules

There are two different Lua initialization modules provided. More could be created for specific purposes. Specify them in the appropriate primary or secondary pragmas (or in the common pragma to apply to both client and server):

linit.c	loads essential modules and the Lua parser. Loads the “propeller” module if the **ENABLE_PROPELLER** Catalina symbol is defined. Loads the propeller and threads modules if **-lthreads** is specified as an option. Usually used in conjunction with the **-llua** option.
xinit.c	loads essential modules, but no Lua parser. Loads the “propeller” module if the **ENABLE_PROPELLER** Catalina symbol is defined. Loads the propeller and threads modules if **-lthreads** is specified as an option. Usually used in conjunction with the **-lluax** option.

Both *linit.c* and *xinit.c* load the **services** module that is used to implement Lua client/servers. This is 'required' by the Lua common module as follows:

```
svc = require 'services'
```

Using the compiled version of the common module is recommended in all Lua client/server scripts, so that it can be loaded by clients and servers that only support compiled Lua programs. For example:

```
dofile 'common.lux'
```

Hub RAM Allocation

The Catapult 'stack' and 'address' attributes are the mechanism used to allocate Hub RAM between the client and the server. Trial and error must be used to determine the appropriate sizes. The minimum stack required by ANY client is about 75,000 bytes (which is the size used by the **eluas** version of the demo).

Note that since the client always executes entirely from Hub RAM, the client stack size (specified as the 'stack' attribute of the secondary catapult pragma) includes the Lua stack and heap. Once this size is determined, the resulting server address (specified as the 'address' attribute of the primary catapult pragma) determines how much Hub RAM is left for the server. However, because the server is an XMM LARGE program, this Hub RAM is used only for the Lua stack and does not include the heap (which is in XMM RAM), so the Hub RAM requirement of Lua code executed in the server is typically much lower than it would be if the same code was executed by the client. Put in simple terms, if there is a choice then Lua code should be put in the server rather than the client..

Here are the approximate amounts of Hub RAM (in bytes) available to the client and the server in each version of eLua:

name	client	server
elua	100,000	100,000
eluax	100,000	90,000
eluas	75,000	180,000
eluasx	100,000	130,000
eluafx	100,000	50,000

Other Hub RAM trade-offs between client and server are of course possible.

Cog allocation

In order to maximize the available free cogs for demonstration purposes, the RTC plugin is not loaded by any of the eLua variants. This may make some Lua functions not work as expected - e.g. **os.date()** or **os.clock()**. If these are required, add the options **-C RTC** (in place of **-C CLOCK**) to the appropriate Catapult pragmas. This requires an additional cog.

The binser module

The **binser**[4] module does not originate with Catalina - the original is available on github[5]. However, Catalina's version has been modified to suit Catalina and Lua 5.4.4, and this version is in the folder *demos/elua/binser*, along with the original documentation.

The **binser** module is normally loaded by the common module, and it is recommended to use the compiled version (*binser.lux*) because it loads faster and will also work in compiled clients and servers (the text version *binser.lua* will not). For example:

```
bs = dofile 'binser.lux'
```

There are only three **binser** functions that are typically required by an **eLua** program:

The function **serialize is used** to turn a series of one or more arguments into a serial binary string (note the result is a Lua string and not a C string - it may contain embedded zeroes):

```
serial = bs.serialize(args ...)
```

For example:

```
serial = bs.serialize(x, y, z)
```

The function **deserializeN** is used to turn a serial binary string back into **N** arguments (if there is only one argument, specify 1 for **N**):

```
arg1, ... argN = bs.deserializeN(serial, N)
```

For example:

```
arg = bs.deserializeN(serial, 1) -- one argument
arg1, arg2 = bs.deserializeN(serial, 2) -- two arguments
```

Finally, there is also a **deserialize** function:

```
args = bs.deserialize(serial)
```

However, **deserialize** returns its results as a Lua table instead of a series of Lua data types, which makes it slightly more complex to use. For instance, instead of using **deserializeN**, to use **deserialize** to do the same job would have to be written as:

```
args = bs.deserialize(serial)
args1 = args[1]
...
argsN = args[N]
```

Using **deserializeN** is easier unless the program *intends* to pass Lua tables, in which case it is easier to use **deserialize**. For instance:

```
args_in = {x=1, y=2} -- this is a table with two elements (x and y)
serial = bs.serialize(args_in)
args_out = bs.deserialize(serial)
```

[4] In all the **eLua** program examples, **binser** is abbreviated to **bs** when loaded.

[5] See https://github.com/bakpakin/binser

There is a Lua subtlety here that can catch you unawares. Both the **deserializeN and deseserialize** functions actually return more than one value, which is a perfectly acceptable thing to do in Lua. The first values returned are the results of deserializing, and the final value is the resulting position in the string being deserialized - something which is generally of little interest in **eLua**.

The subtlety is that if you pass the result of deserializing straight to a function that *accepts* a variable number of arguments - such as the **print** function does - you will get *all* the values.

So, for example:

```
result = bs.deserializeN(bs.serialize("abc"),1)
print(result)
```

will output:

```
abc
```

which is what you would expect. Whereas …

```
print(bs.deserializeN(bs.serialize("abc"),1))
```

which looks like it should do exactly the same thing, will instead output:

```
abc    6
```

which may be a little surprising until you know what is going on!

The ALOHA protocol

The ALOHA protocol is a simple serial protocol specifically designed to be used for client/server transactions.

The protocol is asymmetric. The server never initiates a transaction. The client initiates every transaction by sending a request packet:

```
FF 02 id sq lo hi b1 .. bn ck
```

Where:

`id`	is a service id
`sq`	is a sequence number (can be used to detect repeats)
`lo` and `hi`	are the length of data - i.e. (`hi`<<8)+`lo`
`b1` .. `bn`	are the bytes of data (may contain zeroes)
`ck`	is the checksum of packet (excluding the initial `FF 02`)

The server can respond to the request packet by sending a return `FF 02` packet, which indicates success:

```
FF 02 id sq lo hi b1 .. bn ck
```

Or the server can respond by sending a failure response. Some possible failure responses are:

`FF 01`	timeout on tx
`FF 03`	checksum error on tx
`FF 04`	no such id
`FF 05`	other error

Notes:

- Any occurrence of `FF` other than in the initial `FF 02` is "byte stuffed" to `FF 00`. This means that `FF 02` can never occur within a message, and so it always signals the start of a packet.
- The `ck` is set so that the sum of all the bytes after the `FF 02` equals zero (modulo 0x100).
- The `sq` is not checked by the protocol - it is just a value. Typically it is simply returned by the server in the response to the request. If the service is not idempotent then it is up to the server to ensure it does not service duplicate requests, and it is up to the client to ensure the response packet it receives has the correct `sq`, incrementing it as required to ensure the the response is not simply a duplicate that has been buffered and/or re-transmitted during the serial processing.

The protocol is implemented in the files *aloha.h* and *aloha.c* in *demos/elua/aloha*. This folder also contains a simple demo/test program implemented in the files *amaster.c* and *aslave.c* - see those files for more details.

There are only two functions required to implement the ALOHA protocol. These are described in the sections below.

aloha_tx

Here is the C function prototype of **aloha_tx**, which is used to transmit an ALOHA packet:

```
void aloha_tx(int port, int id, int sq, int len, char *buf);
```

port	0 .. 1 if the 2 port serial plugin is used, 0 .. 7 if the 8 port serial plugin is used.
id	the id (0 .. 255) of the service to be called.
sq	a sequence number that should be incremented on each call. This number will be returned in the response packet so that it is possible for the sender to distinguish a response to a request from a buffered or retransmitted response to a previous request.
len	the length of the message to send.
buf	a pointer to the binary message to send.

aloha_rx

Here is the C prototype of **aloha_rx**, which is used to receive an ALOHA packet:

```
int aloha_rx(int port, int *id, int *sq, int *len, char *buf, int max, int ms);
```

port	0 .. 1 if the 2 port serial plugin is used, 0 .. 7 for the 8 port serial plugin is used.
id	a pointer to an int used to return the id (0 .. 255) of the service that was requested.
sq	a pointer to an int used to return the sequence number of the request.
len	a pointer to an int used to return the size in bytes of the response data.
buf	a pointer to a buffer to put the response data.
max	the size of the buffer.
ms	the timeout in milliseconds to wait for each byte of the response.

The return value is as follows:

0	success (success packet received)
-1	timeout on rx
-2	packet larger than max
-3	checksum error on rx
n	failure packet n received (n != 2)

The ALOHA dispatcher

eLua normally uses the Lua dispatcher which is in the standard Catalina C library and called **_dispatch_Lua_bg**. This dispatcher works perfectly well when the client and server are executing in different cogs on the same propeller - the transfer of data between the client and server can occur in Hub RAM.

However, eLua was designed to be extendable across *multiple* propellers - i.e. when the client is executing on one propeller and one or more servers are executing on another. This is where the ALOHA protocol comes in.

To use the ALOHA protocol, a custom Lua dispatcher is required. This is provided in *dsptch_l.c* in the folder *demos/elua/aloha*. Using this dispatcher requires that the 2 port serial or 8 port serial plugin is used, and that the ALOHA protocol is also included.

This version of the Lua dispatcher has the following additions to the standard Lua dispatcher:

- It retrieves the list of all Lua services (local or remote) from the **service_index** table, and the ports any remote services must use from the **port_0_index** .. **port_n_index** tables (where **n** = 2 or 8, depending on which serial plugin is in use). All services that might need to be dispatched on this propeller must be listed in the **service_index** table. An entry that is listed in the **port_n_index** tables (and it can be listed only in *one* of them) is assumed to be a remote service available on port **n**. Any entry that is listed in the **service_index** table but *not* in any of the **port_n_index** tables is assumed to be a local service.

- ○ It monitors the Catalina Registry as usual for service requests, but if it receives a request for a service whose id corresponds to an id in the **port_n_index** table it sends the request as an ALOHA request out the appropriate serial port, and waits for a response on the same port.Otherwise, it dispatches the service internally (i.e. to the cog indicated in the Catalina Service Registry)
- ○ It monitors all ports which have a **port_n_index** table (even if that table is empty) for the arrival of ALOHA messages. If it receives one, it returns an ALOHA response on the same port.

It is worth noting that ALL services must be included in the **service_index** table, but only services that must be called using ALOHA should be included in the **port_n_index** table.

It is also worth noting that while it is highly recommended that all clients and servers use the same **service_index** table, each client and server may use a different set of **port_n_index** tables. When multiple propellers are configured in a simple star configuration (i.e. with the master propeller connected via a dedicated serial link to each slave propeller) then it is recommended that all propellers use the *same* set of tables (which would typically be specified in **common.lua**). An example of this is given in the demo program. More complex configurations are possible, but are beyond the scope of this document.

The ALOHA dispatcher has the following configurable parameters, which may need to be adjusted for specific applications:

Parameter	Default	Meaning
REMOTE_TIMEOUT	1000	timeout on response after sending request
PORT_TIMEOUT	200	timeout on rx after receiving first byte
REMOTE_MAX	2048	maximum size of message that can be received

Background tasks

The Lua dispatcher executes on each propeller and dispatches calls made via the registry to their appropriate destination (i.e. either to the local slave server, or to a remote slave server). It also continually checks for incoming service calls on the serial ports and dispatches those as well.

But when there are no service calls outstanding, it can also periodically execute a **background** Lua function. It does this between each check of the registry and the nominated serial ports.

The consequence of this is that if the background function adheres to some simple conditions, then it can be used to perform useful tasks. The **background** function must adhere to the following conditions:

1. It cannot accept any arguments or return any values.

2. It must return regularly, so as not to hold up incoming service calls unduly.
3. If ALOHA is in use, it must not execute for long enough on each call to cause an ALOHA timeout - if it executes for a significant portion of the REMOTE_TIMEOUT specified for the ALOHA protocol, then a remote service call may fail with a timeout error.

The function should be written so that on each call, it picks up where it left off from the previous call, does a little more processing, and then returns. Note that it can store state internally between calls.

Here is a trivial example of a background task that flashes an LED periodically:

```
count = 0
LED = 38 -- suitable for P2_EDGE, change to 56 for P2_EVAL
function background()
  count = count + 1
  if count > 1000 then
      propeller.togglepin(LED)
      count = 0
  end
end
```

The **background** function can be defined in **server.lua** for automatic execution by the ALOHA dispatcher on slave servers, or **remote.lua** for automatic execution on the master server.

The background function can also be executed on slave clients, but this is not done automatically. To do it manually, simply add a loop to call **background()** repeatedly. It is recommended to add a small delay between calls, or call **propeller.msleep(0)** between calls (which does a *yield* if multi-threading is in use).

For example, add the following code to **remote_n.lua** for each slave client that should execute the background function:

```
while true do
  background()
  propeller.msleep(0)
end
```

Appendix A - A more complete demo

This appendix contains a more complete and “user friendly” version of the simple demo. This version is in the folder *demos/elua/complete*. Copy the files in this folder to a Catalyst SD card to execute them. Note that they have the same names as the files in the simple version of the demo, and will overwrite them.

No new concepts or features are introduced, but it has comments, does more error checking, has more examples of some features, and the following improvements over the simple version described in the main section of this document:

1. It prints messages when the client and server Lua programs are loaded.
2. It does not require the slave to be executed first. The master will wait for a keystroke before proceeding.
3. It demonstrates both local and remote services, rather than just remote ones (the “quit” service is defined in the “service_index” table in **common.lua**, but not listed in the port_0_index table in **remote.lua** - instead, it is defined in both **server.lua** and **remote.lua**, which means calls to this service will always go to the server local to the client that calls it, not to a remote server).
4. It demonstrates a service (the “big” service) that accepts and returns a string (significantly longer than the size of the buffers used by the serial plugin). Note that to accommodate this, the proxy function for “big” in **common.lua** specifies a larger buffer size (1000 bytes).
5. It demonstrates a means whereby the master can shut down remote slaves (using the “execute” service) as well as local slaves (using the “quit” service).

Execute this program the same way as the simpler version of the demo.

On the master propeller:

master client.lua remote.lua

On the slave propeller:

slave remote.lua server.lua

client.lua

```
print("client ...")

-- load common definitions ...
dofile 'common.lux'

print("... loaded")

function ENTER_to_continue()
  print("\nPress ENTER to continue");
  io.read();
end

-- call some server functions ...

ENTER_to_continue()
print("calling invoke ...")
invoke(function(str) print(str) end, "ALOHA")
```

```
ENTER_to_continue()
print("calling invoke ...")
function f(x)
   return x*x
end
print ("Result = ".. invoke(f, 2.5))

ENTER_to_continue()
print("calling big ...")
input = "Now is the time for all good men to come to the aid of the party
... The quick brown fox jumps over the lazy dog ... Four score and ten
years ago ... we will fight them on the beaches ... hey ho, hey ho - it's
off to work we go ...  "
print ("Result = ".. big(input .. input .. input .. input))

ENTER_to_continue()
print("calling execute (will time out!)...")
execute("")
print("calling quit ...")
quit()
```

server.lua

```
print("server ...")

-- load common definitions ...
common = dofile('common.lux')

-- define remote services (on port 0) - an empty table indicates the
-- server calls no services on the port but just monitors the port
-- for calls from a remote client ...

port_0_index = { }

-- this is the function the server executes in the background ...

function background()
  -- the default is to do nothing!
end

-- define local services ...

-- this service returns the output of invoking function f on value x ...
function invoke(serial)
   f, x = bs.deserializeN(serial, 2);
   output = f(x);
   return bs.serialize(output)
end

-- this service demonstrates accepting and returning a big string ...
function big(serial)
   input = bs.deserializeN(serial, 1);
   output = "bigger ... " .. input
   return bs.serialize(output)
end

-- this service executes a Catalyst command, which will also
-- terminate the slave ...
function execute(serial)
  command = bs.deserializeN(serial, 1);
```

```
    if type(command) == "string" then
        print("Client requested execute '" .. command .. "'")
        propeller.execute(command)
    else
        result = "Invalid execute command";
    end
     return bs.serialize(output)
end

-- this service quits the local server ...
function quit()
  print("Client requested shutdown\n")
  os.exit()
end

print("... loaded")
```

common.lua

```
-- this file must be loaded by both the client and the server. It is the
-- only file required to know about the service ids.
--
-- for example, to load the text version:
--
--   dofile 'common.lua'
--
-- or to load the compiled version:
--
--   dofile 'common.lux'
--
-- using the compiled version is recommended even in text Lua files,
-- so that the files do not have to be modified to be executed using a
-- client/server program that does not load a Lua parser.

svc = require 'service'
bs  = dofile 'binser.lux'

-- define a unique service id for each service:

INVOKE_SVC        = 81
QUIT_SVC          = 82
BIG_SVC           = 83
EXECUTE_SVC       = 84

-- define the services implemented by the server ...

service_index = {
  [INVOKE_SVC]    = "invoke",
  [QUIT_SVC]      = "quit",
  [BIG_SVC]       = "big",
  [EXECUTE_SVC]   = "execute",
}


-- define proxy calls for the services implemented by the server. The
-- value of 500 in these definitions is the maximum size (in bytes) that
-- is expected when the parameters are serialized:

function invoke(f, x)
  out = bs.deserializeN(svc.serial(INVOKE_SVC, bs.serialize(f, x), 500),
1)
```

```
  return out
end

function big(input)
  out = bs.deserializeN(svc.serial(BIG_SVC, bs.serialize(input), 1000), 1)
  return out
end

-- note that this function will ALWAYS generate a timeout if it
-- succeeds, because it shuts the slave down to execute the command:
function execute(input)
  out = bs.deserializeN(svc.serial(EXECUTE_SVC, bs.serialize(input),
1000), 1)
  return out
end

function quit()
  svc.serial(QUIT_SVC, "", 0)
  return nil
end
```

remote.lua

```
print("remote ...")

-- load common definitions ...
common = dofile('common.lux')

-- define remote services (on port 0) ...

port_0_index = {
  [INVOKE_SVC]   = "invoke",
  [BIG_SVC]       = "big",
  [EXECUTE_SVC]  = "execute",
}

-- this is the function the server executes in the background ...

function background()
  -- the default is to do nothing!
end

-- define local services ...

-- this service quits the local server ...
function quit()
  print("Client requested shutdown\n")
  os.exit()
end

print("... loaded")
```

Appendix B - Life

This appendix contains details of a more complex **eLua**/ALOHA client/server demo.

The folder *demos/elua/life* contains two Lua implementations of Conway's Game of Life[6] - a simple cellular automaton.

First, the version of the program in *life.lua* is *not* a client/server program. It is a normal Lua program and it can be compiled and executed using normal Lua. For example:

```
luac -o life.lux life.lua
xl_luax life.lux
```

When executed[7], and using a serial HMI, each iteration of Life takes about 10 seconds.

Another version of the Game of Life is in the files *common.lua*, *client.lua* and *server.lua* (and their compiled equivalents *common.lux*, *client.lux* and *server.lux*). This is a client/server version of the same program. The server implements the cellular automaton, and the client handles the display of the output. Execute it using **eluax**:

```
eluax
```

When executed this way, each iteration of Life takes about 5 seconds - i.e. it is about TWICE AS FAST as the original version of the program. Since the program is now being executed across two cogs instead of one, this is about what we should expect - and with Catalyst, Catapult and Lua it is now possible to do this without much effort.

And with ALOHA, the client and server do not even need to execute on the same propeller.

To demonstrate this, two propellers are needed, prepared as a P2_MASTER and P2_SLAVE, as described in the main body of this document.

On the P2_SLAVE[8], execute **slave** but specify **remote.lua** as the *client*:

```
slave remote.lua server.lua
```

Then, on the P2_MASTER, execute **master** but specify **remote.lua** as the *server*:

```
master client.lua remote.lua
```

Using ALOHA doesn't speed the program up any further. But it would enable much larger Life universes to be created than would be possible without **eLua**.

As Mr Spock might have said to Captain Kirk …

"It's Life, Jim, but not as we know it"

[6] see https://en.wikipedia.org/wiki/Conway's_Game_of_Life

[7] To execute *life.lux* from XMM RAM for this comparison, it must be executed using **xl_luax**, and not **luax**.

[8] The commands *must* be executed in this order - i.e. the slave has to be started *before* the master, or the master will time out.