Who is Who in the Lua Zoo

Table of Contents

Introduction..... 2
Rationale..... 2
Standalone Lua..... 4
Lua : Standard Lua (lua and luax)..... 4
iLua : Interactive Lua (ilua)..... 5
mLua : Multiprocessing Lua (mlua and mluax)..... 5
wLua : WiFi Lua (wlua and wluax)..... 6
Client/Server Lua..... 6
eLua : Extended Lua (elua, eluax and eluaXX)..... 6
aLua : ALOHA/Serial Lua (alua, aluax and aluaXX)..... 7
rLua : RPC/WiFi Lua (rlua, rluax and rluaXX)..... 8
sLua : Server-only Lua (sluaXXX)..... 10
Other variants..... 11

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 – i.e. C programs can call Lua programs and vice-versa. But Lua can also be used stand-alone. This means it is not necessary to either *know* or *use* any C to create and execute Lua programs.

Catalina has extensive Lua support - in fact, the propeller 2 versions of Catalyst and the self-hosted version of the Catalina compiler itself are partly *implemented* in Lua. But this document is not about Lua itself - for that, see any Lua documentation (such as the Lua Reference Manual - see https://www.lua.org/manual/5.4/). Instead, this document gives a brief overview of the different *variants* of Lua supported by Catalina. Further detail of each of the variants is then provided in other Catalina documents:

- The Standard[2] and Interactive variants of Lua are included as Catalyst demo programs. Details are given in the **Catalyst Reference Manual**.
- Details of the Multiprocessing variants of Lua are given in the document **Lua on the Propeller 2 with Catalina**. This document also describes the **propeller** and **hmi** modules which (if enabled) provide basic propeller and HMI functions to all variants.
- Details on the Client/Server variants of Lua, where a Lua Client and a Lua Server execute either on the *same* propeller or on *different* propellers, are given in the document **Aloha from Lua**.

Rationale

Lua is a highly extensible language. While all the Lua variants support the same Lua language, they are extended by including various C or Lua library modules or components. Since the propeller has no support for dynamically loading libraries and has limited Hub RAM, the different variants are required to allow different capabilities to be supported by having different modules compiled into the Lua executable, or omitted to save space. The different variants may also use different memory models (to trade-off between speed and space), different platform configurations and different Catalina plugins and options.

Here is a summary of possible differences between Lua variants:

1. Whether the variant includes the Lua parser. This is a standard Lua option. The main reason for omitting the parser is to save memory if it is not required at run-time. If the parser *is* included, then Lua can execute both source (e.g. *.lua*) files as well as compiled (e.g. *.lux*) files. Otherwise only compiled files can be executed.

---

[1] The term "embedded" here means embedded within other programs, not embedded in hardware.
[2] Standard Lua is supported on the propeller 1 and 2, the other variants only on the propeller 2.

2. The Catalina plugins and Catalina compile-time options used by the variant, such as the cache size to use for variants that use XMM RAM.

3. Whether various propeller-specific library modules and other Lua components are compiled into the variant (to save space if they are not required). These are:

   propeller basic propeller functions (e.g. *setpin(), sleep(), lockset()* etc)

   hmi HMI functions (*k_get(), t_char(), m_button()* etc)

   threads multi-threading functions *(factory(), worker(), channel()* etc)

   service registry service calls (*long(), short(), float(), serial()* etc)

   serial serial protocol functions (*tx(), rx(), txflush(), str()* etc)

   wifi WiFi functions (*SEND(), RECV(), POLL(), CONNECT()* etc)

   linenoise command-line editing support (used by Interactive Lua)

   debug A standard Lua module which provides debug and reflection support

   utf8 A standard Lua module which provides utf8 character support

   math A standard Lua module which provides floating point support

   os A standard Lua module which provides operating system support

   coroutine A standard Lua module which provides co-routine support

4. For the Standalone variants, Lua can be compiled in **NATIVE** mode, which executes entirely from the Hub RAM at maximum speed, **COMPACT** mode, which executes entirely from Hub RAM using less memory but more slowly, or **SMALL** or **LARGE** mode[3], which executes from XMM RAM more slowly still, but which can support Lua programs up to 16Mb.

5. For Client/Server variants, the memory model used by the Client and the Server (which can be different). Clients always use Hub RAM for all code and data. but can use **NMM**[4], **LMM** or **CMM**. Some variants allow the Servers to use either Hub RAM for all code and data, or use XMM RAM for code and Hub RAM for data and stack space (i.e. **XMM SMALL**) or XMM RAM for code and data and Hub RAM only for stack space (i.e. **XMM LARGE**).

6. For Client/Server variants that execute both the Client and the Server on the *same* propeller, the amount of RAM allocated to the Client and the Server.

7. For Client/Server variants where the Client and the Server execute on *different* propellers, the platform configuration to use for each. These are referred to as the **master** and **slave** propellers, and are typically compiled using pairs of platform

[3] The Multiprocessing variants cannot use XMM SMALL or XMM LARGE modes - this is one reason for having the Client/Server variants, described later.

[4] CMM, LMM, NMM and XMM are different memory/execution models for code on the Propeller 2. See the **Catalina Reference Manual (Propeller 2)** for details.

configurations such as **P2_MASTER** and **P2_SLAVE**, or **P2_WIFI_MASTER** and **P2_WIFI_SLAVE** - although note that there could also be *multiple* slaves, with each slave using a different platform configuration.

Standalone Lua

The Standalone variants of Lua are those where there is only one instance of Lua executing. However, that single instance may have several Lua coroutines or threads executing simultaneously, and these may be executing on multiple cogs.

Lua : Standard Lua (lua and luax)

There are two main variants of standard Lua in the folder *demos/catalyst/lua5.4.8*.

Here is a brief description of the two main **lua** variants, available on any propeller 2:

lua	a CMM variant of Lua, *with* a Lua parser.
luax	a CMM variant of Lua *without* a Lua parser.

There are also **lua** variants built to use XMM RAM (if available):

xs_lua	an XMM SMALL variant of Lua *with* a Lua parser.
xs_luax	an XMM SMALL variant of Lua *without* a Lua parser.
xl_lua	an XMM LARGE variant of Lua *with* a Lua parser.
xl_luax	an XMM LARGE variant of Lua *without* a Lua parser.

All variants include the **propeller** and **hmi** modules, and also the **linenoise** module unless the Catalina symbol **NO_LINENOISE** was specified when Catalyst was compiled.

Lua can be invoked directly from the Catalyst command line, optionally specifying the name of the Lua file to execute (specifying a file is *required* for **luax**). For example:

```
lua
lua progam.lua
xs_lua progam.lux
xl_luax program.lux
```

Catalyst can also execute Lua commands directly from the command prompt, so entering just **program** would cause Catalyst to search for (and execute if found) *program.lux* and then *program.lua*. Also, the Catalyst environment variables **LUA** and **LUAX** can be set to specify *which* variant is used (the defaults are to use **lua** and **luax**). For example:

```
set LUA=xs_lua
set LUAX=xl_luax
```

Also in *demos/catalyst/lua5.4.8* is the Lua compiler (**luac**), which is suitable for compiling all Lua programs no matter what Lua variant is used to execute them. Also, in the folder *demos/catalyst/core* is a Lua program *clua.lua* that makes the Lua compiler easier to use (for instance, it is not necessary to specify the extension and output files, as it is for **luac**).

So the command:

```
clua program
```

does the same as:

```
luac -o program.lux program.lua
```

If XMM RAM is available, there are also XMM SMALL and XMM_LARGE variants of the Lua compiler (**xs_luac** and **xl_luac**). To have the **clua** command use these instead of the CMM version, edit the script *(*on the Catalyst SD, this script will be in *bin/clua.lua)*.

Examples of **Lua** programs are given in *demos/catalyst/lua5.4.8/test*.

iLua : Interactive Lua (ilua)

Interactive Lua is not really a separate variant - it is a Lua script (*ilua.lua*) in *demos/catalyst/ilua* that can be executed using Standard Lua. This script provides an enhanced Lua Read-Eval-Print Loop and enhanced command-line editing. It requires that Stand-alone Lua has been built to include the **linenoise** module (which is the default).

Since it is a Lua script, **iLua** can be invoked directly from the Catalyst command line. The script does not accept any parameters. So just enter the command:

```
ilua
```

mLua : Multiprocessing Lua (mlua and mluax)

There are several variants of Lua with multiprocessing support enabled, also in the folder *demos/catalyst/lua5.4.8*. The multiprocessing support allows Lua to execute on multiple cogs.

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

mlua	a CMM variant of Lua with multiprocessing support and *with* a Lua parser.
mluax	a CMM variant of Lua with multiprocessing support *without* a Lua parser.

These variants include the **propeller, hmi** and **threads** modules, and also the **linenoise** module unless the Catalina symbol **NO_LINENOISE** was specified when Catalyst was compiled. To free up more Hub RAM, the **debug** and **utf8** Lua modules are omitted from these variants[5].

mLua can be invoked directly from the Catalyst command line, optionally specifying the name of the Lua file to execute (specifying a file is *required* for **mluax**). For example:

```
mlua
mlua progam.lua
```

[5] If the **debug** or **utf8** modules are required by a Lua program, they can be re-enabled by editing the appropriate Lua initialization file (i.e. *linit.c*). However, this will reduce the size of Lua programs that can be executed.


```
mlua progam.lux
mluax program.lux
```

The multiprocessing extensions to Lua are described in detail in the document **Lua on the Propeller 2 with Catalina**. Examples of **mLua** programs are described in that document, and included in *demos/catalyst/lua5.4.8/test*.

wLua : WiFi Lua (wlua and wluax)

There are two variants of Lua with WiFi support enabled in the folder *demos/wifi*. The WiFi support allows internet applications (e.g. that use **HTTP**, **TCP** or **TELNET**) to be created in Lua.

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

wlua a CMM variant of Lua with WiFi support enabled and *with* a Lua parser.

wluax a CMM variant of Lua with WiFi support enabled *without* a Lua parser.

These variants include the **propeller, hmi, serial** and **wifi** modules. The **linenoise** module is *not* included.

wLua can be invoked directly from the Catalyst command line, optionally specifying the name of the Lua file to execute (specifying a file is *required* for **wluax**). For example:

```
wlua
wlua progam.lua
wlua progam.lux
wluax program.lux
```

Examples of **wLua** programs are included in *demos/wifi*.

Client/Server Lua

The Client/Server variants of Lua allows Lua programs to use a simple Client/Server paradigm, where the Client executes from Hub RAM at maximum speed, and the Server executes from XMM RAM, and can therefore be up to 16Mb in size. The Client and Server can be on the same propeller, or on different propellers.

eLua : Extended Lua (elua, eluax and eluaXX)

The Extended variant of Lua allows Lua programs to use a simple Client/Server paradigm, where both Client and Server execute on the same propeller.

There are several Extended variants in the folder *demos/elua* - two main ones, plus a few others that are useful in particular circumstances.

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

elua a CMM Lua client with Lua multiprocessing support, and an XMM LARGE Lua server with an 8K cache, both *with* a Lua parser.

eluax a CMM Lua client with Lua multiprocessing support, and an XMM LARGE Lua server with a 64K cache, both *without* a Lua parser.

The other **eLua** variants omit the multiprocessing support, which frees up more 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 multiprocessing 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 programs 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 multiprocessing support. This variant also omits the **propeller** module from the Lua Client (it is enabled in the Lua Server).

eluasx a CMM Lua client and an XMM LARGE Lua server with a 64K cache, both *without* a Lua parser or multiprocessing support.

Unless otherwise stated above, the **propeller** and **hmi** modules are included in both the Client and the Server.

eLua can be invoked directly from the Catalyst command line, optionally specifying the names of the Client and Server files to execute. If no extension is specified, then *.lua* is assumed by variants that include the parser, and *.lux* is assumed by variants that do not. If no filenames are entered, then **client** and **server** are assumed. For example:

```
elua
eluax
elua client server
eluax client serverbg
```

eLua is described in detail in the document **Aloha from Lua**. This document also describes the structure required for Lua Client/Server programs. Examples of **eLua** programs are included in *demos/elua*.

aLua : ALOHA/Serial Lua (alua, aluax and aluaXX)

The ALOHA/Serial variant of Lua allows Lua programs to use a simple Client/Server paradigm, where the Client and Server are on different propellers, connected via a hardwired serial connection.

As with **eLua**, the **aLua** variants differ in the memory models and components included. Here is a brief description of the **aLua** variants provided in the folder *demos/elua/aoha*:

alua a CMM Lua client with Lua multiprocessing support, and an XMM LARGE Lua server with an 8K cache, both *with* a Lua parser.

aluax a CMM Lua client with Lua multiprocessing support, and an XMM LARGE Lua server with a 64K cache, both *without* a Lua parser.

aluafx an NMM Lua client and an XMM LARGE Lua server with an 8K cache, both *without* a Lua parser or multiprocessing support. Because the client is a NATIVE program it will execute faster than the other aLua 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.

The **build_aloha** script in the folder *demos/elua/aloha* builds these variants to execute on both **P2_MASTER** and **P2_SLAVE** propellers, and it renames the resulting **P2_MASTER** binaries as *master.bin*, *masterx.bin* and *masterfx.bin*, and the **P2_SLAVE** binaries as *slave.bin*, *slavex.bin* and *slavefx.bin* - so it ends up the following six binaries:

master.bin	**alua** compiled for P2_MASTER
masterx.bin	**aluax** compiled for P2_MASTER
masterfx.bin	**aluafx** compiled for P2_MASTER
slave.bin	**alua** compiled for P2_SLAVE
slavex.bin	**aluax** compiled for P2_SLAVE
slavefx.bin	**aluafx** compiled for P2_SLAVE

Then **master** (or **masterx** or **masterfx**) must be executed on the **P2_MASTER** propeller, and **slave** (or **slavex** or **slavefx**) must be executed on the **P2_SLAVE** propeller. The **master** and **slave** names are arbitrary - in fact all the binaries are simply aLua, but built using a different platform configuration file.

aLua programs can be invoked directly from the Catalyst command line, optionally specifying the names of the Client and Server files to execute. If no extension is specified, then *.lua* is assumed by variants that include the parser, and *.lux* is assumed by variants that do not. If no filenames are entered, then **client** and **server** are assumed. For example:

```
master
slavex
master client server
slave client serverbg
```

aLua is described in detail in the document **Aloha from Lua**. This document also describes the structure required for Lua Client/Server programs. Examples of **aLua** programs are included in *demos/elua/aloha*.

rLua : RPC/WiFi Lua (rlua, rluax and rluaXX)

The RPC/WiFi variant of Lua allows Lua programs to use a simple Client/Server paradigm, where the Client and Server are on the same propeller, or on different propellers, connected to a WiFi network.

As with the **eLua** and **aLua** variants, the **rLua** variants differ in the memory models and components included. Here is a brief description of the **rLua** variants provided:

rlua a CMM Lua client with Lua multiprocessing support, and an XMM LARGE Lua server with an 8K cache, both *with* a Lua parser. Uses the 8 port serial plugin and supports both serial ALOHA and WiFi services. Both the client and the server support the WiFi functions.

rluax a CMM Lua client with Lua multiprocessing support, and an XMM LARGE Lua server with an 8K cache, both *without* a Lua parser. Uses the 8 port serial plugin and supports both serial ALOHA and WiFi services. Both the client and the server support the WiFi functions.

rlua2 a CMM Lua client with Lua multiprocessing support, and an XMM LARGE Lua server with an 8K cache, both *with* a Lua parser. Uses the 2 port serial plugin and does ***NOT*** support serial ALOHA services - only WiFi services, and only the server supports the WiFi functions[6]. This allows more Hub RAM for Lua programs.

rlua2x a CMM Lua client with Lua multiprocessing support, and an XMM LARGE Lua server with an 8K cache, both *without* a Lua parser. Uses the 2 port serial plugin but does ***NOT*** support serial ALOHA services - only WiFi services, and only the server supports the WiFi functions[7]. This allows more Hub RAM for Lua programs.

rluafx an NMM Lua client with Lua multiprocessing support, and an XMM LARGE Lua server with an 8K cache, both *without* a Lua parser. Uses the 2 port serial plugin but does ***NOT*** support serial ALOHA services - only WiFi services, and only the server supports the WiFi functions[8].

The **build_rpc** script in the folder *demos/elua/aloha* builds these variants for the propeller platform **P2_WIFI** to execute on a single propeller or multiple propellers with the same platform configuration, but it also builds them for **P2_WIFI_MASTER** and **P2_WIFI_SLAVE** to be executed on multiple propellers that have different platform configurations. It renames the resulting **P2_WIFI_MASTER** binaries as *rmaster.bin* and *rmasterx.bin* and and the **P2_WIFI_SLAVE** binaries as *rslave.bin* and *rslavex.bin* - to end up the following binaries:

rlua.bin	**rlua** compiled for P2_WIFI
rluax.bin	**rluax** compiled for P2_WIFI
rluafx.bin	**rluafx** compiled for P2_WIFI
rlua2.bin	**rlua2** compiled for P2_WIFI
rlua2x.bin	**rlua2x** compiled for P2_WIFI
rmaster.bin	**rlua** compiled for P2_WIFI_MASTER
rmasterx.bin	**rluax** compiled for P2_WIFI_MASTER
rslave.bin	**rlua** compiled for P2_WIFI_SLAVE
rslavex.bin	**rluax** compiled for P2_WIFI_SLAVE

The **rlua** (or **rluax** or **rluafx** or **rlua2** or **rlua2x**) can be executed on any **P2_WIFI** propeller. The **rmaster** (or **rmasterx**) must be executed on the **P2_WIFI_MASTER** propeller, and **rslave** (or **rslavex**) must be executed on the **P2_WIFI_SLAVE** propeller.

---

[6] However, the client can use the WiFi functions provided by the server if suitable proxy services are defined - see the HTTP example in *demos/elua/http*.

[7] See previous footnote.

[8] See previous footnote.

The names are arbitrary - in fact all the binaries are simply rLua, but built for a specific platform, and including a custom Lua dispatcher with ALOHA serial and WiFi RPC support (except for **rlua2, rlua2x** and **rluafx**, which explicitly disable the ALOHA serial protocol to free up more Hub RAM).

rLua programs can be invoked directly from the Catalyst command line, optionally specifying the names of the Client and Server files to execute. If no extension is specified, then *.lua* is assumed by variants that include the parser, and *.lux* is assumed by variants that do not. If no filenames are entered, then **client** and **server** are assumed. For example:

```
rlua
rluax
rlua client remote
rluax remote serverbg
```

rLua is described in detail in the document **Aloha from Lua**. This document also describes the structure required for Lua Client/Server programs. Examples of **rLua** programs are included in *demos/elua/aloha*.

sLua : Server-only Lua (sluaXXX)

The Server-only Lua variants contain a Lua Server, but no Lua Client (and so they *must* be executed in conjunction with a Client executing on another propeller). This allows the Lua Server to be executed entirely from Hub RAM. Since Clients *always* execute from Hub RAM, this allows for maximum speed where the Lua program can be written as a Client/Server program to execute on multiple propellers.

The following variants are intended to be used with an existing **aLua** client, and are therefore built only for the **P2_SLAVE** propeller:

sluafx an NMM Lua Server with no Lua client, no Lua parser and no multiprocessing support. Because the server is a NATIVE program it will execute faster than some other sLua variants, but only support smaller Lua servers. This variant also omits the **propeller** module.

sluafix an NMM Lua Server with no Lua client, no Lua parser and no multiprocessing support. Because the server is a NATIVE program it will execute faster than some other sLua variants, but only support smaller Lua servers. It uses the integer version of the standard C library, which omits the capabilities of doing I/O on floating point numbers, and also the Lua **math** and **os** modules[9].This variant also omits the **propeller** module.

sluacx a CMM Lua Server with no Lua client, no Lua parser and no multiprocessing support. Because the server is a COMPACT program it will execute more slowly than the **sluafx** and **sluafix** variants, but can support larger Lua servers. This variant also omits the **propeller** module.

[9] If the **math** or **os** modules are required by a Lua program, they can be re-enabled by editing the appropriate Lua initialization file *iinit.c*

sluacix a CMM Lua Server with no Lua client, no Lua parser and no multiprocessing support. Because the server is a COMPACT program it will execute more slowly than the **sluafx** and **sluafix** variants, but can support larger Lua servers. It uses an integer version of the standard C library, which omits the capabilities of doing I/O on floating point numbers, and also the Lua **math** and **os** modules. This variant also omits the **propeller** module to free up more Hub RAM.

The following variant is intended to be used with an existing **rLua** client, and is only built for the **P2_WIFI_SLAVE** propeller:

sluarfx an NMM Lua Server with no Lua client, no Lua parser and no multiprocessing support. Because the server is a NATIVE program it will execute faster than some other sLua variants, but it can only support smaller Lua servers.

Unless otherwise stated above, the **propeller** and **hmi** modules are included in the Lua Server.

sLua programs can be invoked directly from the Catalyst command line, optionally specifying the name of the Lua Server file to execute. If no extension is specified, then *.lua* is assumed by variants that include the parser, and *.lux* is assumed by variants that do not. If no filenames are entered, then **server** is assumed. For example:

```
sluafx
sluacix server
sluafx server.lux
```

sLua is described in detail in the document **Aloha from Lua**. Examples of Server-only Lua programs are included in *demos/elua/aloha*.

Other variants

Note that not all possible Lua variants are included with Catalina - there are simply too many possible combinations. Additional variants to those described in this document may be included in particular Catalina releases, or can be created as needed for specific applications.

Creating new variants can be done by:

1. Editing the relevant Lua initialization files (typically this will be called *linit.c, xinit.c* or *iinit.c*) to enable or disable specific Lua modules.

2. Creating and adding new Lua modules.

3. Editing the platform configuration files in the *target/p2* directory (e.g *P2MASTER.inc*, *P2SLAVE.inc*, *P2WIFI.inc*, *P2WIFI_S.inc*, *P2WIFI_M.inc*) or creating additional ones (e.g. if there is more than one Lua Slave propeller).

4. Editing the build scripts or Makefiles (e.g. to modify command line options, or include additional options such as **NO_LINENOISE** or **ENABLE_PROPELLER**).

5. Editing the catapult pragmas in the relevant C source files (the Catalina **catapult** utility is used to build all the Client/Server Lua variants). See the document **Getting Started with Catapult** for details.

