EPICS lua

Aug 10, 2022

lua Module

Contents

The lua EPICS Module is an embedding of the lua language interpreter into an EPICS IOC. From there, the interpreter is exposed to the user in the form of the luascript record, which uses the interpreter to allow for scriptable record support; the lua shell, an addition to or replacement of the ioc shell; and functions to allow easy use of lua in other modules.

Currently, the lua module uses lua version 5.4.0. A reference manual describing the details of the language can be found here.

lua Module Release Notes

CHAPTER 1

lua Release Notes

1.1 Release 3-0-2

- Various bug fixes relating to proper link handling
- Fixed another windows build issue relating to order of includes
- Enum input field support added, fields are grabbed as string value.

1.2 Release 3-0-1

- Fix issues for windows builds.

1.3 Release 3-0

- Lua language version updated to 5.4.0
- "db" library added. Allows users to generate records within lua scripts during IOC startup.
- luaPortDriver added. Generates an asynPortDriver based off of a lua script. Each parameter is defined within lua and links to a snippet of lua code, with the code being run whenever the parameter is read or written (depending on how the parameter is defined).
- Named lua states added. Can create lua_States that can be shared between instances of lua scripts by name.
- The command exit in the lua shell changed from a specially recognized word to a lua function so that it can be properly parsed within a chunk. Allows it to be used in a conditional or loop.
- luascriptRecord AA-JJ inputs can now take in arrays and the record can also write out to array records.
- luaShell.h API changed, luashBegin renamed to luashLoad to match IOC shell naming conventions, C++ overloads of luash command to allow for lua_States to be given to shell to set the environment.

- Added functions to luaEpics.h to provide scoped environment variables, changed luaLoadMacros to use scopes.
- Fixed an issue with too many temp files being created and deleted by the iocsh library.

1.4 Release 2-1

- LUA_SCRIPT_PATH now always includes the current directory. Makes more sense when using '<' in the lua shell.
- Previously, the "iocsh" library was used only to lookup ioc shell functions, now it will now also check for environment variables that match the given name.
- "iocsh" library lookups now also fixed to return nil when it can't find a matching element (in R2-0 it was returning functions that, when called, stated nothing was found).
- Added setOption function to the "asyn" library. works the same way as asynSetOption. The asyn.client class received a matching function.
- Fixed bug in "asyn" library where writeread requests were attempting to read twice, causing timeout waits.
- Added luashCmd function to ioc shell. Useful for running one-liners of lua code.
- lua shell now specially recognizes the line '#ENABLE_HASH_COMMENTS', when put into a lua shell script, the shell will ignore lines where the first non-whitespace line is a '#' character. Allowing scripts to appear more like regular ioc shell scripts.
- lua shell now ignores leading whitespace on lines, was only an issue with the 'exit' and '<' commands.
- Fixed an issue where I was leaving a metatable reference on the lua stack when luaCreateState was called.
- Documentation has been switched to use ReStructured text, now hosted on https://epics-lua.readthedocs.io/en/latest/

1.5 Release 2-0

- "iocsh" library now available for any version of base
- Calls to the "iocsh" library can omit the library name while within the luash interpreter, this makes the luash almost fully backwards compatible with iocsh scripts. The only problems come from comments (due to "#" being a command in lua) and macros being different than global variables.
- Better error handling in "asyn" library
- lua Script file location now allows full paths
- loadRegistered function automatically triggered and no longer needed for lua startup scripts
- Lua static library registration now setup to work with standard lua "require" functionality
- 'asyn' lua library function "port" changed to "client" to better represent that it is creating an asynOctetClient not an asynPortDriver. InTerminator and OutTerminator changed to member fields rather than get/set functions.
- 'asyn' lua library new function "driver" creates an object representing an asynPortDriver. Allows you to get/set the value of parameters in the paramList and trigger the read and write functions of parameters.
- Parameters supplied in luaRecord CODE field and macros provided to luaSpawn are now evaluated using a lua sandbox environment rather than a custom parser.

1.6 Release 1-3

- Fixes compilation issues for Visual Studio 2010
- Dynamic library loading enabled for Linux architectures
- C functions can now be registered into lua libraries that will automatically load on lua shell startup.
- Also set up a quick way to bind functions already loaded into the IOC shell
- All libraries loaded into shell, device support, and luascript record
- luaEpics functions are now able to be used in C files
- Added luaSpawn function to allow for running scripts in the background
- Full Documentation

1.7 Release 1-2-2

- Fixed a compilation bug on vxWorks
- First official synApps release

1.8 Release 1-2-1

- Fixed a bug where softChannel support wasn't working
- luascript's CODE field is no longer saved by autosave on every luascript record, just for the user luascripts.

1.9 Release 1-2

- Fixed building for Windows
- Added global reference to the pdbbase variable, allows lua shell to be used as a full replacement for iocsh
- Added '<' command to luash to include the contents of other scripts
- luaScript input fields now can have descriptions of their contents
- Fixed a bug where forward link processing wasn't happening
-

1.10 Release 1-1

- luash function now runs as a command shell or an interpreter
- Lua asyn library function to access ports as lua objects
- LUASH_PS1 variable for lua shell prompt

1.11 Release 1-0

- First public release on github

Suggestions and Comments to:
Keenan Lang: (klang@aps.anl.gov)

CHAPTER 2

luascriptRecord

The luascript record type provides customizeable record behavior in much the same way as the calcout or scalcout record. Every time the record is processeed, lua code is executed and any returned variables are outputted to an OUT_LINK. The record has both double and string input links that get exposed as global variables for the code to use.

The record has access to a set of *library functions* that allow it to get and put values from other PV's or asyn port parameters, sleep, call iocsh functions, send commands to a device, or you can *bind your own functions* to extend its functionality.

Full documentation can be found *here*

2.1 luascript - Lua Script Processing Record

Contents

2.1.1 Introduction

The lua script processing record or "luascript" record is derived from the Calcout record in EPICS base, but replaces the calc operation engine with the lua scripting language. The record has 10 string fields (AA...JJ) and 10 double fields (A...J) whose values are retrieved every time the record is processed and those values are pushed into lua as global variables with the same name as the field.

The luascript record has both a VAL and SVAL output field. If the return operator is used within a lua expression, the returned value is placed into one of these fields. Booleans or Numbers that are returned get their value put into the VAL field, while Strings will be put into the SVAL field.

When writing to a string PV (any of DBF_STRING, DBF_ENUM, DBF_MENU, DBF_DEVICE, DBF_INLINK, DBF_OUTLINK, DBF_FWDLINK) the record (actually, device support) writes its string value (SVAL). When writing to any other kind of PV, the record writes its numeric value (VAL).

To write successfully to a DBF_MENU or DBF_ENUM (for example, the VAL field of a `bo` or `mbbo` record) the record's string value must be one of the possible strings for the PV, or it must an integer specifying the string number [0..N] for the PV. For example, when writing to a `bo` record whose ZNAM is "No" and whose ONAM is "Yes", the string value must be one of the following: "No", "Yes", "0", or "1". To ensure that numeric values are converted to integers, set the precision (the PREC field) to zero.

2.1.2 Scan Parameters

The luascript record has the standard fields for specifying under what circumstances the record will be processed. These fields are listed in Scan Fields, Chapter 2, 2. In addition, Scanning Specification, Chapter 1, 1, explains how these fields are used. Since the luascript record supports no direct interfaces to hardware, it cannot be scanned on I/O interrupt, so its SCAN field cannot be `I/O Intr`.

2.1.3 Read Parameters

The read parameters for the luascript record consist of 20 input links: 10 to numeric fields (INPA -> A, INPB -> B, ... INPJ -> J); and 10 to non-numeric fields (INAA -> AA, INBB -> BB, ... INJJ -> JJ). The fields can be database links, channel access links, or constants. If they are links, they must specify another record's field. If they are constants, they will be initialized with the value they are configured with and can be changed via `dbPuts`. Non-numeric input links check the field type of the link provided and fetch data as either strings or as an array of values.

In addition, the luascript record contains the fields INAV, INBV, ... INJV, which indicate the status of the links to numeric fields, and the fields IAAV, IBBV, ... IJJV, which indicate the status of the links to string fields. These fields indicate whether or not the specified PV was found and a link to it established. See *Section 5, Operator Display Parameters* for an explanation of these fields.

See the EPICS Record Reference Manual for information on how to specify database links.

Field	Summary	Type	DCT	Initial	Access	Modify	Rec Proc Monitor
INPA	Input Link A	INLINK	Yes	0	Yes	Yes	N/A
INPB	Input Link B	INLINK	Yes	0	Yes	Yes	N/A
...	...	...	...	...	...	...	...
INPL	Input Link J	INLINK	Yes	0	Yes	Yes	N/A
INAA	Input Link AA	INLINK	Yes	0	Yes	Yes	N/A
INBB	Input Link BB	INLINK	Yes	0	Yes	Yes	N/A
...	...	...	...	...	...	...	...
INJJ	Input Link JJ	INLINK	Yes	0	Yes	Yes	N/A

2.1.4 Expressions

The luascript record has a CODE field into which you can enter an expression for the record to evaluate when it processes. The return operator can be used to return either a numeric or string variable for writing to the VAL or SVAL field respectively. Either VAL and SVAL can also be written to the output link. (If you elect to write an output value, the record will choose between VAL and SVAL, depending on the data type of the field at the other end of the output link.)

The CODE expression can also be used to reference a file containing the description of at least a single lua function. If the CODE field starts with the symbol '@' followed by the name of said file, the luascript record will search through a list of directories given by the environment variable 'LUA_SCRIPT_PATH' (default: current directory) for the given file. A space character and then the name of a function defined in the file lets the luascript record know what function to call when the record processes. Optionally, a set of parameters can be provided that the function will be called with each processing by including a comma separated list enclosed by parentheses.

When changing the CODE field, the luascript record's RELO field controls whether or not the record will recompile the string into a new lua state, resetting any variables in the global scope. The field is a menu with three choices:

- `Every New File` – Recompile only if the file referenced is changed, the record can be changed to point to a new function within that file without losing any prior state.
- `Every New Change` – Recompile on any change to the CODE field.
- `Every Processing` – Recompile before each time the record is processed.

There is also the FRLD field which forces the record to recompile a new lua state when a non-zero value is written to it.

Finally, the ERR field contains a string representation of the last error encountered during processing.

The record also has a second set of calculation-related fields described in *Section 4, Output Parameters.*

Field	Summary	Type	DCT	Initial	Access	Modify	Rec Proc Monitor	PP
CODE	Script	STRING[120]	Yes	0	Yes	Yes	Yes	No
VAL	Value	DOUBLE	No	0	Yes	Yes	Yes	No
SVAL	String value	STRING (40)	No	0	Yes	Yes	Yes	No
RELO	When to reload state?	Menu	Yes	0	Yes	Yes	No	No
FRLD	Force Reload	Short	Yes	0	Yes	Yes	No	No
ERR	Last Error	String (200)	No	0	Yes	Yes	No	No

Examples

field(CODE, “return A + B”)

- Sets VAL to the result of A + B

field(CODE, “return AA .. BB”)

- Sets SVAL to the concatenation of AA and BB

field(CODE, “@test.lua example”)

- Runs the function ‘example’ from the file test.lua with zero parameters.

field(CODE, “@test.lua example(1, ‘foo’)”)

- Runs the function ‘example’ from the file test.lua with two parameters, one a number, the other a string.

2.1.5 Output Parameters

These parameters specify and control the output capabilities of the luascript record. They determine when to write the output, where to write it, and what the output will be. The OUT link specifies the Process Variable to which the result will be written. The OOPT field determines the condition that causes the output link to be written to. It’s a menu field that has six choices:

- `Every Time` – write output every time record is processed.
- `On Change` – write output every time VAL/SVAL/AVAL changes, i.e., every time the result of the expression changes to a value different than the one immediately previous.
- `When Zero` – when record is processed, write output if VAL is zero, if SVAL is an empty string, or if AVAL is a 0-sized array.
- `When Non-zero` – when record is processed, write output if VAL is non-zero, SVAL is a non-empty string, or if AVAL has any elements.
- `Transition to Zero` – when record is processed, write output only if VAL is zero and last value was non-zero. If SVAL was changed, write output only if SVAL is an empty string and the last value was a non-empty string. If AVAL was changed, write output only if AVAL has zero elements and the last array had at least one element.
- `Transition to Non-zero` – when record is processed, write output only if VAL is non-zero and last value was zero. If SVAL was changed, write output only if SVAL is a non-empty string and the last value was a empty string. If AVAL was changed, write output only if AVAL has at least one element and the last value had no elements.

- `Never` – Don't write output ever.

The SYNC field controls whether the record processes in a synchronous or asynchronous manner. It is a menu field with two choices:

- `Sync` – process the record's lua code synchronously.
- `Async` – process the record's lua code in a separate thread.

Field	Summary	Type	DCT	Initial	Access	Modify	Rec Proc Monitor	PP
OUT	Output Specification	OUTLINK	Yes	0	Yes	Yes	N/A	No
OOPT	Output Execute Option	Menu	Yes	0	Yes	Yes	No	No
SYNC	Synchronicity	Menu	Yes	0	Yes	Yes	No	No

The luascript record uses device support to write to the `OUT` link. Soft device supplied with the record is selected with the .dbd specification

```
field(DTYP,"Soft Channel")
```

2.1.6 Operator Display Parameters

These parameters are used to present meaningful data to the operator. Some are also meant to represent the status of the record at run-time. An example of an interactive MEDM display screen that displays the status of the luascript record is located here.

The HOPR and LOPR fields only refer to the limits of the VAL, HIHI, HIGH, LOW, and LOLO fields. PREC controls the precision of the VAL field.

The INAV-INJV and IAAV-IJJV fields indicate the status of the link to the PVs specified in the INPA-INPJ and INAA-INJJ fields, respectively. The fields can have three possible values:

Ext PV NC	the PV wasn't found on this IOC and a Channel Access link hasn't been established.
Ext PV OK	the PV wasn't found on this IOC and a Channel Access link has been established.
Local PV	the PV was found on this IOC.
Constant	the corresponding link field is a constant.

The OUTV field indicates the status of the OUT link. It has the same possible values as the INAV-INJV fields.

See the EPICS Record Reference Manual, for more on the record name (NAME) and description (DESC) fields.

Field	Summary	Type	DCT	Initial	Access	Modify	Rec Proc Monitor	PP
PREC	Display Precision	SHORT	Yes	0	Yes	Yes	No	No
HOPR	High Operating Range	FLOAT	Yes	0	Yes	Yes	No	No
LOPR	Low Operating Range	FLOAT	Yes	0	Yes	Yes	No	No
INAV	Link Status of INPA	Menu	No	1	Yes	No	No	No
INBV	Link Status of INPB	Menu	No	1	Yes	No	No	No
…	…	…	…	…	…	…	…	…
INJV	Link Status of INPJ	Menu	No	1	Yes	No	No	No
OUTV	OUT PV Status	Menu	No	0	Yes	No	No	No
NAME	Record Name	STRING [29]	Yes	0	Yes	No	No	No
DESC	Description	STRING [29]	Yes	Null	Yes	Yes	No	No
IAAV	Link Status of INAA	Menu	No	1	Yes	No	No	No
IBBV	Link Status of INBB	Menu	No	1	Yes	No	No	No
…	…	…	…	…	…	…	…	…
IJJV	Link Status of INJJ	Menu	No	1	Yes	No	No	No

2.1.7 Alarm Parameters

The possible alarm conditions for the luascript record are the SCAN, READ, Calculation, and limit alarms. The SCAN and READ alarms are called by the record support routines. The Calculation alarm is called by the record processing routine when the CALC expression is an invalid one, upon which an error message is generated.

The following alarm parameters which are configured by the user define the limit alarms for the VAL field and the severity corresponding to those conditions.

The HYST field defines an alarm deadband for each limit. See the EPICS Record Reference Manual for a complete explanation of alarms and these fields.

Field	Summary	Type	DCT	Initial	Access	Modify	Rec Proc Monitor	PP
HIHI	Hihi Alarm Limit	FLOAT	Yes	0	Yes	Yes	No	Yes
HIGH	High Alarm Limit	FLOAT	Yes	0	Yes	Yes	No	Yes
LOW	Low Alarm Limit	FLOAT	Yes	0	Yes	Yes	No	Yes
LOLO	Lolo Alarm Limit	FLOAT	Yes	0	Yes	Yes	No	Yes
HHSV	Severity for a Hihi Alarm	Menu	Yes	0	Yes	Yes	No	Yes
HSV	Severity for a High Alarm	Menu	Yes	0	Yes	Yes	No	Yes
LSV	Severity for a Low Alarm	Menu	Yes	0	Yes	Yes	No	Yes
LLSV	Severity for a Lolo Alarm	Menu	Yes	0	Yes	Yes	No	Yes
HYST	Alarm Deadband	DOUBLE	Yes	0	Yes	Yes	No	No

2.1.8 Monitor Parameters

These parameters are used to determine when to send monitors for the value fields. The monitors are sent when the value field exceeds the last monitored field by the appropriate deadband, the ADEL for archiver monitors and the MDEL field for all other types of monitors. If these fields have a value of zero, every time the value changes, monitors are triggered; if they have a value of -1, every time the record is scanned, monitors are triggered.

Field	Summary	Type	DCT	Initial	Access	Modify	Rec Proc Monitor	PP
ADEL	Archive Deadband	DOUBLE	Yes	0	Yes	Yes	No	No
MDEL	Monitor, i.e. value change, Deadband	DOUBLE	Yes	0	Yes	Yes	No	No

2.1.9 Run-time Parameters

These fields are not configurable using a configuration tool and none are modifiable at run-time. They are used to process the record.

2.1.10 Record Support Routines

init_record

For each constant input link, the corresponding value field is initialized with the constant value if the input link is CONSTANT or a channel access link is created if the input link is PV_LINK.

The CODE field is processed and either compiled into bytecode directly, or the record will search for a given file and compile that file into bytecode.

process

See section 11.

special

This is called if CODE is changed.

get_precision

Retrieves PREC.

2.1.11 Record Processing

process()

The `process()` routine implements the following algorithm:

1. Recompile the CODE field if the RELO field is set to "Every Processing".

2. Push the values of all input links to global lua variables.

3. Run the compiled code in a separate thread. Process the returned value from the code to determine if it is a numeric value or a string value. Update VAL or SVAL accordingly.

4. Determine if the Output Execution Option (OOPT) is met. If it is met, execute the output link (and output event).

5. Check to see if monitors should be invoked.

Monitors for A-J and AA-JJ are set whenever values are changed.

6. Set PACT FALSE.

CHAPTER 3

luaPortDriver

luaPortDriver support is included to generate asynPortDrivers with parameters generated from a lua script. Each parameter gets snippets of lua code associated with its reading and writing that get called when the asyn callbacks are triggered.

Full documentation can be found *here*

3.1 luaPortDriver

Within the lua module support is the lua and IOC shell function luaPortDriver. This function takes in the name of an asyn port, a lua script, and a string of macro definitions. An example of a luaPortDriver function call is shown below:

```
luaPortDriver("EXAMPLE", "exampleDriver.lua", "VAL=10")
```

An asynPortDriver is created with the given asyn port name and the lua script is run with the defined macro values. Within the script, parameters can be implemented using the following convention:

```
param.<param_type> "<NAME>"
```

The parameter type can be any of Int32, Float64, or Octet, each corresponding to the equivalent asynParamType that shares their name. You can also use String as a alternative for an Octet definition. None of these definitions are case sensitive. Name defines the name that the parameter is created with.

The above, basic definition of a parameter only creates a correctly typed parameter in the port driver. Values can be written or read from the parameter, but nothing else is actually done. Instead, a slightly more advanced form is used to bind lua code to reading and writing callbacks.

```
param.<param_type>.<read/write> "NAME" [[
   CODE
]]
```

The same parameter type and name conventions remain, but the definition now also includes a specifier on whether you are providing a function for the reading of a parameter or the writing of a parameter. The code is then lua code as a multi-line string.

Implicitly defined within the code is a variable named "self". This is a lua driver object as defined in the asyn library and represents the luaPortDriver you are creating. This is useful for being able to read from and write to other parameters in the driver during execution. As well, for code that implements a write callback, you also have the variable "value" that contains the value to write coming from the asyn callback. For read callbacks, a value that is returned by the code will be written out to the value parameter in the asyn callback.

Put together, here is a small example of a functional luaPortDriver for a simple calculation of the length of a hypotenuse:

```
param.int32 "BASE"
param.int32 "SIDE"

param.float64.read "HYPOTENUSE" [[
   return math.sqrt(self["BASE"]^2 + self["SIDE"]^2)
]]
```

CHAPTER 4

lua Shell

The lua shell is an alternate shell to either the vxWorks or ioc shell environment. It has all the same functionality of those shells, while also providing the ability to conditionally execute code, calculate necessary values, construct loops over code, and define functions within startup scripts.

The lua Shell is able to access all the same *epics lua libraries* as the luascriptRecord, including any functions or libraries that you *build yourself* into your IOC. The shell even implicitly loads the iocsh library, which means you can call iocsh-registered functions in the lua shell exactly like you would in the ioc shell.

Further Information

4.1 Using the Lua Shell

The lua shell is exposed as both a c function and is registered as a function with iocsh. Thus, the shell can either be invoked in a startup script or be run as the startup program in general.

The shell has been set up so as to be as backwards compatible with the iocsh style startup scripts as possible. While within the lua shell, the global environment is set up so that lookups of names that don't have an associated variable will attempt to pull values from, first, the running epics environment, and then, if no environment variable is found, try to find a matching function name. This means that functions that are registered with the ioc shell can be treated as if they were defined lua functions.

```
luash> EPICS_VERSION_MAJOR
7
luash>
luash> epicsEnvShow
func_meta: 0x673150
luash>
luash> epicsEnvShow("EPICS_VERSION_MAJOR")
EPICS_VERSION_MAJOR=7
```

As well, the special directive '#ENABLE_HASH_COMMENTS' is provided. While lua normally reserves the '#' character for determining the length of a table, putting the line '#ENABLE_HASH_COMMENTS' in your scripts

will set the shell to accept iocsh style comments that use '#'. This setting only applies to lines where '#' is the first non-empty character in the line, it will not affect the use of '#' in normal lua operations.

```
luash> #ENABLE_HASH_COMMENTS
Accepting iocsh-style comments
luash>
luash> #print("This won't print")
luash> print(#"Check len")
9
```

4.1.1 Calling the Lua Shell From Inside The IOC Shell

Once the dbd has been loaded and the registerRecordDeviceDriver command has been called on an IOC, you can call the *luash* command. Luash takes in two parameters, the first, the name of a lua script to run, and the second a set of macros to be set as global variables in the lua shell's state. The macros are defined in the same "aaa=bbb,ccc=ddd" form that the ioc shell uses for iocshLoad/iocshRun and that is used for dbLoadRecords/dbLoadTemplate. One difference, however, is that strings need to be quoted in order to be recognized as string variables, any non-quoted value will be interpreted as a number. So to create a global numeric variable X with the value 5 and a string variable TEXT with the value "Hello, World!" you would have your second parameter as "X=5,TEXT='Hello,World'".

The luash command will then attempt to find the script you have given it. It does this by searching for a file with the given name in a set of folders as given by the environment variable LUA_SCRIPT_PATH. If this variable is undefined, the command will search within the current directory. As well, if no script name is given, the shell will take commands from the standard input, with a prompt set by the environment variable LUASH_PS1. Additionally, within the shell, you can find and include other scripts by using the '<' character. Putting

```
< script.lua
```

Will attempt to find script.lua by the same process as detailed above and include all the text of that file at the insertion point. When running any script the lua shell will exit at the end of the file being read.

For both files and commands from the standard input, a single line with only the word 'exit' will break from the current level of shell execution. In example, if you were to load the file script.lua as seen above, and the file had the following code within it

```
if (true) then
    exit
end
```

That exit will end the reading of just the script.lua file and flow would resume to the shell that called script.lua.

As a note, the use of 'exit' is replaced automatically with an actual call to a function 'exit()' by the shell. This is done to make the code proper in regards to lua syntax, while still being able to provide syntactic sugar.

4.1.2 Lua Shell As A Replacement For The IOC Shell

The lua shell can also be called as a c function, which allows it to be used on vxWorks or as a replacement for the IOC shell in a soft IOC. The same parameters apply as when calling the command inside the IOC shell. So your main.cpp file might look like:

:: #include "luaShell.h"

int main(int argc,char *argv[]) {

if(argc>=2) { luash(argv[1]); epicsThreadSleep(.2);

} luash(NULL); epicsExit(0); return(0);

}

4.1.3 Common Lua Environments

In the above code, there are two different lua shell calls. One to read in a given script, and one to provide an interactive shell. As the code stands, any objects that are created in the input script aren't available to the interactive shell. Say, for example, you have a library that provides useful functions that are used in scripts, you would have to include that library again once you are in the interactive shell, and there wouldn't be a way to automate that.

Therefore, there is a way to define that you want different calls to luash to use a common environment. The luashSetCommonState command is included in the luaShell.h file and sets a default environment that subsequent luash calls will use. The command takes a c-string, if the string matches the name of an environment created using luaNamedState(), it will use that environment, otherwise it will create a new state with that name. So the above code changed to allow the interactive shell to reference code from the interpreted script would look like:

:: #include "luaShell.h"

int main(int argc,char *argv[]) {

luashSetCommonState("default");

if(argc>=2) { luash(argv[1]); epicsThreadSleep(.2);

} luash(NULL); epicsExit(0); return(0);

}

4.2 Included lua Library Functions

In addition to the standard lua libraries, the following are additional libraries built into the lua runtime interpreter to help with common epics tasks.

The 'asyn' Library - *Documentation* - This library contains functions to allow users to get and set asyn parameters and communicate over an asyn octet port. It is primarily for use as an easy debugging tool for devices that have a command-response style control scheme, or a framework to allow control of said devices.

The 'db' Library - *Documentation* - This library contains functions to allow users to generate database records like one would using the dbLoadDatabase function to load a db file. Instead, all records can be generated entirely within lua.

The 'epics' Library - *Documentation* - This library contains functions to get and set pv values, as well as operating system independent tasks, like letting a thread sleep.

The 'iocsh' Library - *Documentation* - This library provides the tools to interact with the existing epics framework for the ioc shell. It's major use is to allow the lua shell to be able to call all the same functions as the ioc shell without having to have to change the code for those functions.

4.2.1 Asyn Library Documentation

Getting / Setting Parameters

- **asyn.getParam** (*portName[, addr], paramName*)
- **asyn.getStringParam** (*portName[, addr], paramName*)
- **asyn.getDoubleParam** (*portName[, addr], paramName*)

- **asyn.getIntegerParam** (*portName[, addr], paramName*)

```
Fetches the value of an asyn parameter. These work like the asynPortDriver
functions of the same name, retrieving the value from the param list.


portName    [string] - The registered asyn port name that contains
                       the parameter you are getting.

addr        [number] - The asyn address of the parameter. Optional,
                       default value is 0.

paramName   [string] - The name of the parameter to fetch.


Returns the value of the asyn parameter as the type specified, if no
type was specified, uses the asynParamType of the parameter to determine
```

- **asyn.setParam** (*portName[, addr], paramName*)
- **asyn.setStringParam** (*portName[, addr], paramName, value*)
- **asyn.setDoubleParam** (*portName[, addr], paramName, value*)
- **asyn.setIntegerParam** (*portName[, addr], paramName, value*)

```
Sets the value of an asyn parameter. These work like the asynPortDriver
functions of the same name, saving the value in the param list.


portName    [string] - The registered asyn port name that contains
                       the parameter you are setting.

addr        [number] - The asyn address of the parameter. Optional,
                       default value is 0.

paramName   [string] - The name of the parameter to set.

value       [varies] - The value to set the parameter to. Type should
                       match the type of the parameter you are setting.
```

asyn.callParamCallbacks (*portName[, addr, parameter]*)

```
Tells an asyn port to call parameter callbacks on changed values.


portName    [string] - A registered asyn port name.

addr        [number] - The index of the parameter list to do callbacks on. Optional,
                       default value is 0.

parameter   [string] - A specific parameter to do callbacks on. Optional,
                       default is to perform callbacks on all values that have
                       been changed.
```

Reading / Writing Values

asyn.readParam (*portName[, addr], paramName*)

```
Calls the read function of the correct asyn interface

portName    [string] - The registered asyn port name that contains
                       the parameter you are getting.

addr        [number] - The asyn address of the parameter. Optional,
                       default value is 0.

paramName   [string] - The name of the parameter to fetch.


Returns the value of the asyn parameter as the type specified, if no
type was specified, uses the asynParamType of the parameter to determine
```

asyn.writeParam (*portName[, addr], paramName, value*)

```
Calls the write function of the correct asyn interface


portName    [string] - The registered asyn port name that contains
                       the parameter you are setting.

addr        [number] - The asyn address of the parameter. Optional,
                       default value is 0.

paramName   [string] - The name of the parameter to set.

value       [varies] - The value to write to the parameter. Type should
                       match the type of the parameter you are setting.
```

Configuration Parameters

asyn.setOutTerminator (*terminator*)

```
Sets the global variable OutTerminator, which controls asyn write commands


terminator  [string] - The string value to append to the end of all asyn
                       write calls.
```

asyn.getOutTerminator ()

```
Returns the value of the global variable OutTerminator
```

asyn.setInTerminator (*terminator*)

```
Sets the global variable InTerminator, which controls asyn read commands


terminator  [string] - The string value to wait for when reading from an
                       asyn port.
```

asyn.getInTerminator ()

```
Returns the value of the global variable InTerminator
```

asyn.setWriteTimeout (*timeout*)

```
Sets the global variable WriteTimeout, which controls asyn write commands


timeout     [number] - The number of milliseconds for an asyn write command
                       to wait before failure.
```

asyn.getWriteTimeout ()

```
Returns the value of the global variable WriteTimeout
```

asyn.setReadTimeout (*timeout*)

```
Sets the global variable ReadTimeout, which controls asyn read commands


timeout     [number] - The number of milliseconds for an asyn read command
                       to wait before failure.
```

asyn.getReadTimeout ()

```
Returns the value of the global variable ReadTimeout
```

Debug Information

- **asyn.setTrace** (*portName[, addr], key, val*)
- **asyn.setTrace** (*portName[, addr], {key1=val1, ... }*)

```
Turns on or off asyn's tracing for a mask on a given port. Valid keys are
"error", "device", "filter", "driver", "flow", and "warning", case insensitive.


portName    [string]  - A registered asyn port name.

addr        [number]  - The asyn address of the parameter. Optional,
                        default value is 0.

key         [string]  - Which mask to change

val         [boolean] - Whether to turn on or off the mask
```

- **asyn.setTraceIO** (*portName[, addr], key, val*)
- **asyn.setTraceIO** (*portName[, addr], {key1=val1, ... }*)

```
Turns on or off asyn's tracing for a mask on a given port. Valid keys are
"nodata", "ascii", "escape", and "hex", case insensitive.


portName    [string]  - A registered asyn port name.

addr        [number]  - The asyn address of the parameter. Optional,
                        default value is 0.

key         [string]  - Which mask to change
```

(continues on next page)

(continued from previous page)

```
val         [boolean] - Whether to turn on or off the mask
```

Octet Communications

asyn.write (*data, portName[, addr, parameter]*)

```
Write a string to a given asynOctet port


data        [string] - The string to write to the port. This string will
                       automatically have the value of the global variable
                       OutTerminator appended to it.

portName    [string] - A registered asyn port name.

addr        [number] - The asyn address of the parameter. Optional,
                       default value is 0.

parameter   [string] - An asyn parameter to write to. Optional.
```

asyn.read (*portName[, addr, parameter]*)

```
Read a string from a given asynOctet port


portName    [string] - A registered asyn port name.

addr        [number] - The asyn address of the parameter. Optional,
                       default value is 0.

parameter   [string] - An asyn parameter to read from. Optional.


Returns a string containing all data read from the asynOctet port until encountering
the input terminator set by the global variable InTerminator, or until the timeout set
by the global variable ReadTimeout is reached.
```

asyn.writeread (*data, portName[, addr, parameter]*)

```
Writes data to a port and then reads data from that same port.


portName    [string] - A registered asyn port name.

addr        [number] - The asyn address of the parameter. Optional,
                       default value is 0.

parameter   [string] - An asyn parameter to read and write to. Optional.


Returns a string containing all data read from the asynOctet port until encountering
the input terminator set by the global variable InTerminator, or until the timeout set
by the global variable ReadTimeout is reached.
```

asyn.setOption (*portName[, addr], key, val*)

```
Sets driver-specific options


portName    [string] - A registered asyn port name.

addr        [number] - The asyn address of the parameter. Optional,
                       default value is 0.

key         [string] - The name of the option you are setting.

val         [string] - The value to set the option to.


Returns the asynStatus value of the asynSetOption call.
```

asynOctetClient Object

asyn.client (*portName[, addr, parameter]*)

```
Returns a table representing an asynOctetClient object. This object has the functions
read, write, and readwrite, which work the same as the functions above, but the port
and address need not be specified. The client copies the global in and out terminators
at creation, but you can also set the table's InTerminator and/or OutTerminator fields
manually to a different value.


portName    [string] - A registered asyn port name.

addr        [number] - The asyn address of the parameter. Optional,
                       default value is 0.

parameter   [string] - A specific asyn parameter. Optional.
```

- **client:trace** (*key, val*)
- **client:trace** (*{key1=val1, ... }*)

```
Turns on or off asyn's tracing for a given mask on the port this client is connected
to. Valid keys are "error", "device", "filter", "driver", "flow", and "warning", case
insensitive.

key         [string]  - Which mask to change

val         [boolean] - Whether to turn on or off the mask
```

- **client:traceio** (*key, val*)
- **client:traceio** (*{key1=val1, ... }*)

```
Turns on or off asyn's tracing for a given mask on the port this client is connected
to. Valid keys are "nodata", "ascii", "escape", and "hex", case insensitive.

key         [string]  - Which mask to change

val         [boolean] - Whether to turn on or off the mask
```

client:setOption (*key, val*)

```
Sets an asynOption for the port this client is connected to.

key          [string] - The name of the option you are setting.

val          [string] - The value to set the option to.


Returns the asynStatus of the asynSetOption call.
```

asynPortDriver Object

asyn.driver (*portName*)

```
Returns a table representing an asynPortDriver object. You can read to and write to
keys in the table and the table will try to resolve the names as asyn parameters,
calling getParam or setParam as necessary. The table also indexes the addresses that
the asynPortDriver implements, so driver[1]["VAL"] gets the VAL param associated
with address 1, rather than the default 0.

portName     [string] - A registered asynPortDriver port name
```

- **driver:readParam** (*paramName*)
- **driver:writeParam** (*paramName, value*)

```
Calls the read or write function of the correct asyn interface based upon
the asynParamType of the parameter being written to or read from.

paramName    [string] - The name of a parameter in the driver

value        [varies] - The new value to have the driver write (for writeParam)


Returns the value the the driver returns from the read function (for readParam)
```

4.2.2 Database Library Documentation

db.entry ()

```
Creates a DBENTRY pointer which can be used with all implemented static database
access functions. dbFreeEntry will be called on the pointer automatically when the
entry is garbage collected.
```

Static Database Access

The following list of static database access functions are implemented, largely unchanged from their C API. Naming conventions have been changed to drop the initial "db" prefix and for the next character to be lowercase; so, for example, dbGetFieldName would become the module function getFieldName.

Functions which would return a status code, like dbFindRecord, will instead return a boolean representing success or failure (true or false respectively). Where a DBENTRY pointer is required as a parameter, instead instances of the dbentry class will be used, created with the aforementioned db.entry function.

- getNRecordTypes

- findRecordType
- firstRecordType
- nextRecordType
- getRecordTypeName
- getNFields
- firstField
- nextField
- getFieldDbfType
- getFieldName
- getDefault
- getPrompt
- getPromptGroup
- putRecordAttribute
- getRecordAttribute
- getNAliases
- getNRecords
- findRecord
- firstRecord
- nextRecord
- getRecordName
- isAlias
- createRecord
- createAlias
- deleteRecord
- deleteAliases
- copyRecord
- findField
- foundField
- getString
- putString
- isDefaultValue
- getNMenuChoices
- getMenuIndex
- putMenuIndex
- getMenuStringFromIndex
- getMenuIndexFromString

- getNLinks
- getLinkField
- firstInfo
- nextInfo
- findInfo
- getInfoName
- getInfoString
- putInfoString
- putInfo
- deleteInfo
- getInfo

db.registerDatabaseHook (dbhook)

```
Registers the provided function so that it is invoked each time the dbLoadRecords
function is called by the IOC. The callback hook is invoked with two parameters;
the first being the filepath to the database file being loaded, and the other
being a table of the macro definitions provided to dbLoadRecords.

dbhook    [function]  - The callback function to be invoked
```

db.record ([recordtype,] recordname)

```
Creates an instance of the dbrecord class, a wrapper around record creation/access.

recordtype   [string] - The typename of the record (ai, mbbo, calc, etc) Optional.
                        If the typename is left out, constructor will operate only
                        to find a record, not create one.

recordname   [string] - The name of the record. If the name already exists, the
                        returned instance will refer to the existing record. If
                        there is no record by that name, the constructor will
                        create one.

Returns a class instance with four instance methods, name, type, field, and info.
'name' and 'type' are accessor methods that will return the record name and the
RTYP of the record.

'field' and 'info' are both functions that take in two strings as parameters, the
first being a name and the second a value. 'field' attempts to find the record
field with the given name and then calls dbPutString to set the value. While 'info'
calls dbPutInfo to add a new info field with the given name and value to the record.


rec = db.record("stringin", "x:y:z")
rec:field("VAL", "test")
rec:info("autosave", "VAL")


The class instance itself can also be called as a function, taking in a dictionary
of name-vale pairs. In doing so, the 'field' function is called for each pair,
passing through the names and values to the function.
```

(continues on next page)

(continued from previous page)

```
With lua syntactical sugar, you can chain together the record creation and the
setting of fields like so:

db.record("ai", "x:y:z") {
   DTYP = "asynInt32",
   INP = "@asyn(A_PORT,0,1)PARAM_NAME"
}
```

db.list ()

```
Returns a list of all the PVs currently defined in the IOC. Each element of the
list is a db.record instance.
```

4.2.3 Epics Library Documentation

epics.get (*PV*[, timeout])

```
Calls ca_get to retrieve the value of a PV accessible by the host.


PV          [string] - The name of the PV to request.

timeout     [number] - Amount of seconds to search for pv before giving a timeout,
                       default is 1.0 (can be fractional).


Returns the value of the PV given or Nil if the PV cannot be reached.
```

epics.put (*PV, value*)

```
Calls ca_put to set the value of a PV accessible by the host.


PV          [string] - The name of the PV to request.

value       [varies] - The new value you want to set the PV to. The type of this
                       parameter should match with the dbtype of the PV requested.
```

epics.sleep (*seconds*)

```
Tells the epics thread running the lua script to sleep for a given time.


seconds     [number] - Amount of seconds to sleep for (can be fractional).
```

epics.pv (*PV*)

```
Returns a table representing a PV object. Index accesses can be used to retrive or
change record fields. These changes are completed through ca_get or ca_put.

PV          [string] - The name of the PV to request.
```

4.2.4 IOC Shell Functions

iocsh. <item> [(*arguments...*)]

```
Performs an environment check for the given item. First, the user's environment is
checked to see if there are any environment variables that match up with the item's
name. If there aren't, the search then drops back to trying to find a matching
ioc shell function. Only functions that are registered in the iocsh function database
will be found. If there are no matching elements in either of these two locations,
a nil will be returned.

When using the lua shell interpreter, this functionality is embedded into the global
environment. Any attempt to reference a name that hasn't been set as a lua variable
will attempt a search to see if the name references an environment variable or iocsh
function. These are, however, read-only accesses. If you attempt to set a given
item name to a value, all you will do is create a new lua variable with the given
value.

arguments   [varies] - If you are referencing an ioc shell function, these are the
                       arguments that will get sent to the function. Since function
                       references can be passed around, parentheses are necessary to
                       actually invoke the function.
```

4.3 Adding Libraries to the lua Environment

4.3.1 Dynamic Libraries

If you are on a system that supports dynamic libraries, you can add new functions into lua using the *require* function. Compile a dynamic library with the function int luaopen_xxxx(lua_State* L), where xxxx matches the name of the library file.

Then, you'll need to tell lua the location of the library in question. This is accomplished by either setting the environment variable LUA_CPATH or LUA_CPATH_5_3 before lua is invoked, or the global variable package.cpath if you are already in lua.

These variables are not a set of search directories, like a normal path, however. Instead, they are a set of templates that just replace a wildcard character with the library name you are looking for and check to see if that file exists. So, a package.cpath that is set as

```
"./?.so;/usr/local/?/init.so"
```

would try to search for the files ./foo.so and /usr/local/foo/init.so when you call require("foo"). The first such file that is found gets dynamically loaded and then the function luaopen_xxxx is attempted to be called, in this instance luaopen_foo (**NOTE:** If you compile the library with epics, the resulting file will be called libxxxx.so or libxxxx.dll, so you should include the lib part in the search path).

That luaopen function is then where you would bind your functions into the lua_state. Lua provides the luaL_newlib function to make this easy. You just provide it with a list of pairs of function names and function pointers like so:

```
static int l_bar( lua_State *L )
{
    lua_pushstring(L, "Hello, World");
    return 1;
}
```

(continues on next page)

(continued from previous page)

```
int luaopen_foo( lua_State *L )
{
    static const luaL_Reg foo[] = {
        { "bar", l_bar },
        { NULL, NULL } /* Sentinel item */
    };

    luaL_newlib( L, foo );
    return 1;
}
```

Then you can use the above library like so:

```
foo = require("foo")

print(foo.bar())
```

Note that luaopen_foo just returns a table with all the functions indexed to their correct names, which is why the return value of the require function then needs to be assigned to a variable name.

4.3.2 Static Libraries

Using static libraries is very similar to using dynamic ones. The major difference is that you will have to tell lua what the name of the function that opens your library is rather than giving it a path to find the library. In the luaEpics.h file, there is a function, *luaRegisterLibrary*. The function takes in a name for the library and a function with the same signature as the luaopen_xxxx that we used before. It would be good practice to just continue to use the same luaopen_xxxx naming convention.

The recommended way to get the libraries to be registered correctly for the shell is to use the dbd's registrar function.

foo.cpp

```
static void fooRegister(void)
{
    luaRegisterLibrary("foo", luaopen_foo);
}

extern "C"
{
    epicsExportRegistrar(fooRegister);
}
```

foo.dbd

```
registrar(fooRegister)
```

Then when you load the dbd file into your IOC, lua is given the link between "foo" and the luaopen_foo function, so when you use require to try to load the "foo" library, it will call the open function registered to that name.

Lua will use the built-in means of searching for libraries first, before looking through the libraries registered with *luaRegisterLibrary* so if you have a lua file or shared library in your path or cpath, it will load that rather than the static library.

4.3.3 Adding Individual Functions

You are also able to register individual functions into the global scope, though it isn't necessarily recommended due to possible collisions with existing functions.

To do this, use the *luaRegisterFunction* command pretty much just like you would use the *luaRegisterLibrary* call. Instead of a library name, it takes the name you want the function to have. And instead of taking a function that registers other functions, it just takes the functions directly.

Example:

```
static int l_bar( lua_State *L )
{
    lua_pushstring(L, "Hello, World");
    return 1;
}

static void testRegister(void)
{
    luaRegisterFunction("bar", l_bar);
}
```

Then, you can call the function bar in the lua shell. Since these functions are not part of any library, you don't need to use *require* to load them.