LuaPLC

Introduction

Raritan Inc.

April 17, 2019

Contents

1 LuaPLC Briefly

Lua - A powerful and fast scripting language, simple to learn and use
PLC - Programmable Logic Controller

The LuaPLC feature lets you write small programs and execute them directly an the PDU. Scripts can be deployed using the web GUI, via JSON-RPC or with USB flash drives. They can be started and stopped manually, triggered by event rules or scheduled with timer events.

Lua scripts have full access to the PDU data model as well as a few selected configuration interfaces. Additionally, scripts can use JSON-RPC to control remote PDUs.

2 Quickstart

2.1 Requirements

- A firmware with LuaPLC support (3.3.10 and later)
- The Raritan JSON-RPC SDK

2.2 Upload and Run a Lua Script

1. Download the JSON-RPC SDK and unpack it
2. Open the device web GUI in a browser, login as administrator
3. Navigate to *Device Settings >Lua Scripts*, click *Create New Script*
4. Click *Load Local File* and select *get_info.lua* from the folder *LuaPLC_Examples/basic* in the SDK
5. Select a name for the script, e.g. *get_info*
6. Click *Create* at the bottom of the page
7. On the script status page, click *Start*. The *Script Output* window should now show some information about the PDU.

3 Writing Your Own Scripts

3.1 Lua

Lua is a powerful scripting language, easy to learn and simple to use. See the links section below to find about more the language itself.

3.2 Example Script

Listing 1: Lua example script

```
-- load the "Pdu" module
require "Pdu"

-- acquire a proxy object for the pdumodel.Pdu interface
pdu = pdumodel.Pdu:getDefault()

-- print the PDU model name
metadata = pdu:getMetaData()
print("PDU Model Name: " .. metadata.nameplate.model)

-- print the current sensor reading for each inlet
inlets = pdu:getInlets()
for _, inlet in ipairs(inlets) do
    label = inlet:getMetaData().label
    current_sensor = inlet:getSensors().current
    current = current_sensor:getReading().value
    print("Inlet " .. label .. " Current Reading: " ..
       current .. " A")
end
```

There are a few details worth mentioning in the script above:

- Its first step is to load the *Pdu* module, and all modules it depends on. This module is required to use interfaces from the IDL-defined *pdumodel* namespace.
- The next line uses the static method **pdumodel.Pdu:getDefault()** to acquire a proxy object for the *pdumodel.Pdu* interface. This proxy object can be used to invoke the methods defined in the *Pdu.idl* file. It's the starting point for all scripts working with the PDU data model. All other PDU-related objects (inlets, outlets, sensors, etc.) can be acquired from here.

- Next, the script invokes the **getMetaData()** method on the Pdu proxy object. Note the **:** character between object and method name, it's a critical part of the syntax! As defined in IDL, the call returns a *pdumodel.Pdu.MetaData* structure which includes the model name in the field *nameplate.model*.

- In the following line, the script invokes the **getInlets()** method on the Pdu proxy. The method returns a list of proxy objects for the *pdumodel.Inlet* interface defined in *Inlet.idl*. For a typical PDU with a single inlet, the list contains exactly one item.

- The **for** loop in the next line iterates over the returned inlet proxies. **ipairs** is a Lua-builtin function that enumerates the elements in a list. For each iteration, it returns the index and value of the next list element. Since we're only interested in the value (the inlet proxy), we assign the index to a variable called _.

- For each loop iteration, the variable *inlet* is updated to contain the next inlet proxy from the list. The loop body calls the methods **getMetaData()** and **getSensors()** on this proxy. Again, take note of the **:** syntax to invoke object methods.

- Finally, the script selects the *current* sensor from the returned sensors structure. It contains a proxy for the *sensors.NumericSensor* interface. The script calls **getReading()** on that proxy to retrieve the latest sensor reading and prints the result to the script output.

3.3 IDL to Lua Mapping

Table 1 shows how IDL data types are mapped into Lua.

3.3.1 IDL Files

Each IDL file is mapped to a Lua library that must be loaded with the **require** statement before use. For instance, to use the *Pdu* interface defined in the *Pdu.idl* file, use the statement **require "Pdu"**.

Loading a Lua library automatically loads all directly or indirectly referenced libraries. For instance, requiring the *Pdu* automatically includes many other modules, like *Inlet*, *Outlet*, *NumericSensor* or *PeripheralDeviceManager*.

Table 1: IDL Lua type mapping

IDL	Lua
boolean	boolean
int	number
long	number
float	number
double	number
string	string
time	number
enumeration	number
structure	table
vector	table
map	table
interface	table

3.3.2 Enumerations

An IDL enumeration value is represented by a number in Lua. Additionally, there are constants for each element of the enumeration defined in IDL. For instance, to set the *startupState* field in the *pdumodel.Pdu.Settings* structure can be set to any of the values in the *pdumodel.Pdu.StartupState* table:

Listing 2: Example for using an enumeration variable

```
settings = pdu:getSettings()
settings.startupState =
   pdumodel.Pdu.StartupState.PS_LASTKNOWN
pdu:setSettings(settings)
```

3.3.3 Interface References and Methods

References to IDL interfaces are represented by Lua tables containing the defined methods. To invoke a method, append the method name to the object reference separated by **:** character. IDL *in* parameters are mapped to required function arguments in Lua. Methods return a tuple of IDL-defined return value (unless *void*) and output parameters (if any).

Listing 3: Lua method mapping

```
-- one in parameter, one return value
rc = pdu:setSettings(settings)
if rc ~= 0 then
    print("setSettings failed, rc = " .. rc)
```

```
end

-- no in parameters, three out parameters
inlet, ocp, poles = outlet:getIOP()
```

3.4 Root Objects

Root objects are the entry points for using the IDL-defined API. They are single instances of IDL interfaces that can be acquired using a static **getDefault()** method. References to all other interfaces, like inlets, outlets and sensors, can be reached directly or indirectly from one of these root instances.

The following interfaces contain a static **getDefault()** method to acquire a root instance:

- event.Engine
- luaservice.Manager
- pdumodel.Pdu

3.5 Remote Objects

Remote objects are proxies that communicate with a remote object instance via JSON-RPC. Remote proxies can be created for any supported interface using the static **newRemote()** method. Expected parameters are a resource ID (URL suffix) and an HTTP agent (an object containing a reference to a remote PDU's JSON-RPC service).

The signature to create a new HTTP agent is: **agent.HttpAgent:new(host, user, password [, port [, useTls]])**. Required parameters are a host name or IP address, a user name and a password. Optional parameters are a port number and a boolean flag whether to use HTTP or HTTPS.

Listing 4: Example for using a remote Pdu

```
ha = agent.HttpAgent:new("10.0.42.3", "user", "password")
pdu = pdumodel.Pdu:newRemote("/model/pdu/0", ha)
print("Device Name: " .. pdu:getSettings().name)
```

3.6 Script Arguments

Lua scripts can have arguments that are specified upon startup. Arguments are key-value pairs that can be accessed through the global table **ARGS**. Arguments

can be specified persistently in the script options or using the "Start Script with Arguments" function. Arguments passed at script start override the default arguments from the script options

Listing 5: Get command line arguments

```
paramOutlet = ARGS["outlet"]   -- value for key outlet
paramDelay = ARGS["delay"]     -- value for key delay
```

3.7 Exit Handler

If a script implements a global function named **ExitHandler()**, that function will be executed when the script stops unexpectedly, either because it crashes or is terminated. As a best practice, put this function at the top of the script (right after the require statements) and do not use any global variables within.

Listing 6: Exit handler usage

```
require "Pdu"

-- my special exit handler
function ExitHandler()
    print("Exiting now")
end
```

3.8 Delaying Execution

The built-in sleep function can be used to pause the script for a defined time. The argument is specified in seconds, but fractional numbers are supported for sub-second delays.

Listing 7: Simple Lua sleep function demonstration

```
sleep(2) -- a 2 second pause
```

3.9 Exception Handling

Without precautions, Lua scripts are terminated upon system errors in an IDL model calls (e.g. because using an object reference that doesn't exist). The built-in Lua function **pcall(f [, args...])** can be used to execute function *f* in protected mode. With pcall, the function to be called and its arguments must be specified separately. This includes the reference to the object the method should be invoked on, which must be passed as a first argument. The convenient syntax with **:**

character cannot be used unless the method call is wrapped in an anonymous function:

Listing 8: Correct use of pcall() example

```
-- Correct use of pcall(): Pass object reference explicitly
err, msg = pcall(outlet.setSettings, outlet, s)

-- Correct use of pcall(): Wrap method call in anonymous
   function
err, msg = pcall(function() outlet:setSettings(s) end)

-- The following will NOT work:
err, msg = pcall(outlet:setSettings(s))

-- test if error happens
if err == false then
    print("error caught: " .. msg)
else
    print("no error")
end
```

3.10 Limitations

There are some Lua script limitations:

- Number of deployed scripts
- Script size (per script and total)
- Memory usage per script
- Script CPU utilization

The actual limits can be queried using the **getEnvironment()** method of the **luaservice.Manager** interface.

3.10.1 Out of Memory

Memory usage per script is limited. A script allocating too much memory will be killed by the system. A message like **LuaSvcScript: Out of Memory. Aborting ...** is written to the script output.

The Lua interpreter has a built-in garbage collector. Normally you don't have to worry about allocating or freeing memory. Still, you can manually trigger a

garbage-collection cycle by calling the **collectgarbage()** function. If your script creates a large number of new objects in a short time, running a manual garbage collection cycle may help to prevent a *out of memory* condition.

4 Deploying and Running Scripts

Scripts can be deployed via the following interfaces:

- Web GUI
- JSON-RPC
- USB Flash Drive
- DHCP/TFTP

4.1 Running Scripts from USB Drives

In addition to uploading scripts to the PDU, script files can be executed directly from a USB mass storage device. You need to put the script on a USB drive, along with a control file called (for historical reasons) *fwupdate.cfg*. The control file needs to supply the administrator credentials and a reference to the script file:

```
user=admin
password=raritan
execute_lua_script=my_script.lua
```

The script will be started as soon as the USB drive is plugged into the PDU. Any output will be stored in a separate log file on the drive. If the script is still running by the time the USB drive is disconnected it will be terminated. You can register an ExitHandler, but its runtime is limited to three seconds before the script forcibly is killed.

4.2 Running Scripts from DHCP/TFTP

Running scripts from a TFTP server works similar to the USB drive method. Check the appendix in the PDU User Guide for details about the required configuration of the DHCP and TFTP servers. Runtime for scripts started via this method is limited to 60 seconds. Script output will be written back to the TFTP server, if the server allows it.

4.3 Starting Scripts with Event Rules

Lua scripts can be started or stopped as an action in the event rules engine. Use the web GUI (*Device Settings >Event Rules* to create a new action. Select *Start/Stop Lua Script* as action, then select the script you want to control.

Scripts started by an event rule will receive a number of arguments containing information about the matching event rule and the event triggering it. Arguments are stored in the global table **ARGS**.

5 Links

Some Lua links:

http://www.lua.org
: The offical Lua homepage. Includes a detailed reference manual.

http://www.lua.org/pil/contents.html
: The online version of the book "Programming in Lua" (third edition).

http://tylerneylon.com/a/learn-lua/
: Lern Lua in 15 minutes - more or less. To program with LuaPLC you need to know sections 1, 2 and 3. Sections 3.1 and 3.2 are nice to know.

https://en.wikipedia.org/wiki/Lua_(programming_language)
: Wikipedia article about Lua.