Introduction

Pharos controllers offer many useful show control capabilities. Frequently it is the ability to cope with the particular show control needs of a project that is the critical factor in selecting a control system.

Show control broadly consists of two tasks. First we need to be able to interface with other devices, which may either be triggering us or be under our control. The Pharos LPC supports most of the core interfaces typically used for show control, either directly on the unit (contact closures, RS232, MIDI, TCP/IP, time and date) or via expansion modules or RIOs (RS485, SMPTE). Within the Triggers screen of the Designer software we can configure the LPC to detect particular triggers and how to respond to them.

Second we need to be able to make decisions. These could be simple choices between two alternatives – perhaps a contact closure needs to trigger a different timeline depending on whether it is during the day or during the night. Within the Triggers screen we support a range of conditions that can be used to quickly implement this sort of logical decision making. We also provide a facility to treat values received on an input as a variable that can be used to alter the behaviour of actions – such as using a number received via RS232 to select a particular timeline.

The standard capabilities offered in the Triggers screen are extensive, but a good show control system has the ability to cope with situations that are anything but standard. Within the Pharos system when things get non-standard then we can use scripting.

Script is a simple programming language that allows users to extend the functionality of the Pharos system themselves. We use a freely available programming language called Lua. Anyone who has ever worked with a programming language will find all the typical tools are available, and it should be straightforward to pick up for those who have not. On top of the core Lua syntax we have added some dedicated Pharos functions that allow scripts to work directly with the capabilities of a controller.

Not every problem requires script, but there are few show control problems that can't be solved using script where necessary. A few examples of situations where you might want to use script include:

- Making a single contact closure start a different timeline each time
- Make a timeline loop a set number of times and then release
- Track motion sensor activity over a period of time
- Inverting a DMX input before it is used with a Set Intensity action
- Interpreting data from a wind direction sensor
- Using a table of times for high and low tide to control bridge lighting
- Implementing an interactive game for a science museum

We will use some of the situations as examples below.

The Basics

There are a few basic things you need to know straight away. If any of them are not immediately clear then don't worry – there are lots of examples of how to apply them in the following section.

Lua scripts are written as simple text files using any text editor. It is standard practice to use a .lua filename extension though this is not required. These text files can be loaded directly into the script editor dialog within Pharos Designer.

Comments

It is good practice to include readable comments in your scripts so that you (or anyone else) will be able to easily tell what you were aiming to achieve. In Lua everything after two dashes on a line is treated as a comment.

```
-- This is a comment
This = is + not - a * comment   -- but this is!
```

It is also possible to create multi-line comments using two dashes and two square brackets.

```
--[[
Everything within this section
will be a comment and will have
no effect on the script
--]]
```

The whole point of comments is that they have no effect on the behaviour of the script. But I am introducing them first so that I can use them within the examples that follow.

Variables

If you want to store a piece of data – whether it is a number, some text or just true or false – then you use a variable. You create a variable simply by giving it a name and using it in your script. A variable can store any type of data just by assigning it.

```
firstVariable = 10                      -- assign a number
anotherVariable = "Some text"       -- assign a string
```

When you next use these names then they will have the values that you assigned to them:

```
nextVariable = firstVariable + 5    -- value of
                                    -- nextVariable will
                                    -- be 15
```

Note that names are case-sensitive (i.e. capitals matter!), and once you have named a variable once then anytime you use the same name you will be referring to the same variable – in programming terms it is *global*. This even applies across different scripts – so you can assign a number to a variable called "Bob" in one script and then use the number in another script by referencing "Bob".

One of the most common errors when writing scripts is trying to use a named variable before it has been assigned a value – this will result in an error when the script is run. It is also very easy to use the same name in two different places and not realise that you are actually reusing a single variable. (There is a way of dealing with this for names you want to reuse that we will touch on later.)

Arithmetic

Scripts will often need to do some arithmetic – even if it is something very basic like keeping a counter of how many times it is run:

```
myCount = myCount + 1
```

All of the standard arithmetic operations are available. There is also a library of mathematical functions available should it be required, which includes things like random number generators.

Flow of Control

In most scripts there will be one or more points where you want to make choices. Lua provides four useful structures for this. The most common is “if”, where you can choose which path to take through the script by performing tests.

```
if myNumber < 5 then
  -- first choice
elseif myNumber < 15 and myNumber > 10 then
  -- second choice
else
  -- third choice
end
```

The other control structures all involve blocks of script that need to be repeated a certain number of times. The most straightforward is the “while” loop, which will repeat the enclosed block of script as long as the test at the start is true:

```
myNumber = 10
while myNumber > 0 do
  -- some useful script
  myNumber = myNumber - 1      -- myNumber counts down
end
```

The “repeat-until” loop is really exactly the same, but here the test is done at the end of each loop and it will repeat while the test is false.

```
myNumber = 1
maxNumber = 4096
repeat
  -- some useful script
  myNumber = myNumber * 2
until myNumber == maxNumber
```

Here it is worth noting the use of two equal signs “==” to mean “is equal to” in a test. This is different from a single equal sign, which is used for assigning values. It is another very common mistake to assign a value when you meant to test if it was equal, and it can be hard to spot because it is valid syntax that will not generate an error. The opposite of “==” meaning “is equal to” is “~=” meaning “is not equal to”.

The other control structure is the “for” loop, which has a number of powerful options beyond the scope of what we need here. But it is worth seeing how it can be used to do basic loops in a slightly neater way:

```
for i = 1,10 do
  -- some useful script where i has value 1 to 10
end
```

A final word of caution regarding loops: be careful that you do not write a loop that will never exit! This is all too easy to do by forgetting to increment a counter value that you are using in the test for the loop. If your script has one of these “infinite loops” then the LPC will get stuck when it runs the script and be reset by the watchdog feature (provided this is enabled). Script is a tool for the grown-ups and it will not protect you from doing silly things – so make sure you test your scripts carefully before leaving them to run.

Tables

Often you will need to store a set of values within a script – these might be a list of timeline numbers or the current states of all the contact closure inputs. Lua allows us to store multiple values within a single named variable and this is called a Table.

A table has to be created before it can be used:

```
firstTable = {}                -- creates an empty table
secondTable = { 5,3,9,7 }      -- a table with 4 entries
```

You can then access entries within the table by indexing into it – signified by square brackets. The number within the square brackets identified which entry within the table you want to use or modify.

```
X = secondTable[3]                    -- x now equals 9 (3rd entry)
firstTable[1] = 5              -- entry 1 now has value 5
firstTable[7] = 3              -- entry 7 now has value 3
x = firstTable[1] + firstTable[7]    -- x now equals 5 + 3
```

Note that we are allowed to assign values to entries within the table without doing anything special to change the size of the table. We can keep adding elements to the table as needed and Lua will take care of it for us. This makes it possible to write scripts using tables that will work regardless of how many entries there are in the table (e.g. a list of 4 timeline numbers or of 40).

Tables are particularly powerful when used together with the loops we looked at in the previous section. For example if I have a table of numbers and I wanted to find the smallest then I could use the following script:

```
numbers = { 71,93,22,45,16,33,84 }
smallest = 10000          -- initialise with large number
i = 1                     -- use to count loops
while numbers[i] do
  if numbers[i] < smallest then
    smallest = numbers[i]
  end
  i = i+1
end
```

This is our first really functional piece of script and there are a couple of things worth noting.

- The first entry in a table is accessed using the number one (i.e. myTable[1]). This may seem obvious – but some other programming languages start counting from zero.
- As we increment the variable ‘i’ each time around the loop this means we will be looking at a different entry in the table each time around. The test at the start of my while loop is written to work regardless of how many entries there are in the table. When you use a table entry in a test like this then it will be true as long as the entry has some value (even if the value is zero) and false if there is no value there at all.

Functions

Within script there are a whole range of pre-defined operations that you can call when writing your own scripts. Some of these are provided by the Lua language and are fully described in its documentation. Others have been provided by Pharos to allow you to interact with the LPC from script and are fully described in the manual. They are all called functions and accessed using a similar syntax. For example:

```
x = math.random(1,100)
```

This will assign variable x a value that is a random number between 1 and 100. The function math.random() is a standard function provided by Lua and we can control its behaviour by passing in an argument – in this case the values 1 and 100 to tell it the range within which we want our random number to fall.

```
t = 5
start_timeline(t)
```

This is one of the functions provided by Pharos and it will start the timeline with the number passed in as an argument.

It is also possible to define your own functions as part of script. You might do this if there is a block of script that you know you will need to reuse in a lot of different places. It will be much easier to write the script in one place and then call it from wherever you need it.

```
function diff(a, b)
  if a > b then
    return a - b
  else
    return b - a
  end
end

v1 = 10
v2 = 6
v3 = diff(v1,v2)
```

Note that the script containing the function definition must have been run before we try to call the function. It is often useful to have a script that is run by the LPC startup trigger which defines your functions and creates any tables – other scripts that are run by triggers can make use of those functions and tables.

Practical Examples

In this section we will go through a number of practical examples of how script can be used with an LPC. These examples are all based on real projects that are installed and working. They do get progressively more involved, so do not worry if you don't follow the later ones – you will still be able to use script successfully to solve many problems.

If you are working through this document on your own then look out for where I ask a question and rather than reading straight on I recommend stopping and trying to answer it yourself. You will only get truly comfortable with script by doing it!

Cycling through different timelines

We are installing a wall of RGB LED fixtures in a children's play area. There is a single large button that the kids are supposed to press. Each time they press it they should get a different colour or effect on the wall.

Each colour or effect would be programmed as a different timeline in Designer. The button will connect to a contact closure and so we will have a single Digital Input trigger. Rather than starting a timeline directly we will instead run the following script:

```
-- which timelines should we cycle through?
timeline = { 22, 14, 24, 16, 15, 17, 21 }
n_timeline = 7

-- on first time of running, initialise index
if not index then
  index = 1
end

-- start the timeline whose number is at entry 'index'
get_timeline(timeline[index]):start()

-- increment index
index = index + 1

-- should we go back to the beginning of the table?
if index > n_timeline then
  index = 1
end
```

How would this change if we wanted each button press to choose a timeline at random rather than cycling through them in order?

```
-- which timelines should we cycle through?
timeline = { 22, 14, 24, 16, 15, 17, 21 }
n_timeline = 7

-- use the random function to set index
index = math.random(1,n_timeline)

-- start the timeline whose number is at entry 'index'
get_timeline(timeline[index]):start()
```

Of course if the timeline selection is truly random then it will sometimes select the same timeline twice in a row. If we wanted to prevent this from happening how could we do it?

```
-- which timelines should we cycle through?
timeline = { 22, 14, 24, 16, 15, 17, 21 }
n_timeline = 7

-- find an index different from the old one
while index == oldIndex do
  -- use the random function to set index
  index = math.random(1,n_timeline)
end
```

```
-- store the index for next time round
oldIndex = index

-- start the timeline whose number is at entry 'index'
get_timeline(timeline[index]):start()
```

Stopping a Range of Timelines

We need to stop a large number of timelines in one go, but not all of them.

You can use up to 32 Actions on a Trigger, so if you need to stop more than 32 Timelines at once, you will need to use a script.

You can stop a single timeline from a script with the following:

```
get_timeline(1):stop()
```

This allows you to stop a single timeline from a script, but if you have a large number to stop, adding this for each timeline is a lot of work.

A FOR loop can be used to reduce the amount of scripting required.

```
for i=1,10 do -- run through the values 1-10
    get_timeline(i):stop() -- Stop the timeline defined by i
end
```

This script can be used to run through from 1 to 10 (or a different range by changing the values), and will stop the timeline with those numbers. To make this more useful, you can put it in a function which allows you to call it with any range of timeline numbers.

```
function stop_range(a, b) -- this defines the script as a function
with two variables (a and b)
    for i=a,b do -- a FOR loop which runs through from a to b
        get_timeline(i):stop() -- stop the timeline defined by i
    end
end

stop_range(1,10) -- call the function with the variables 1 and 10
```

Make a timeline loop N times

The designer has requested that a particular timeline runs once at sunset on a Monday, but twice at sunset on a Tuesday, three times at sunset on Wednesday, etc. He is planning to keep changing the timeline so does not want to have lots of copies.

There are actually lots of perfectly reasonable ways to solve this using script. Let's assume we have a single astronomical clock trigger that fires at sunset and runs the following script:

```
N = realtime.weekday    -- 0 is Sunday, 1 is Monday,…

-- we want Sunday to be 7 rather than 0
if N == 0 then
  N = 7
end

get_timeline(1):start()
```

The timeline would be set to loop when it was programmed. We also put a flag on the timeline at the end and make a flag trigger that runs a second script:

```
-- decrement N
N = N - 1

if N == 0 then
  -- release timeline 1 in time 5s
  get_timeline(1):stop(5)
end
```

Note how this works by setting the value of the variable N in one script and then using that variable in another script, which is often a useful technique.

I have used two scripts here, but it is possible to do the same job using only one – can you see how?

In this case you would have the sunset trigger start the timeline directly and use the following script on the flag trigger:

```
-- is this the first time round?
if not N or N == 0 then
  N = realtime.weekday  -- 0 is Sunday, 1 is Monday,…

  -- we want Sunday to be 7 rather than 0
  if N == 0 then
    N = 7
  end
end

-- decrement N
N = N - 1

if N == 0 then
  inject_trigger(2)     -- runs action on trigger 2
end
```

The trick here is to detect whether it is the first time round the loop – if the Controller has started up today then N will have no value and so “not N” will be true, otherwise N will have been left with the value zero when the script ran yesterday. When we detect it is the first time then we set its initial value in the same way as before.

I have also used a different method to do the timeline release. Rather than calling stop_timeline directly from the script I am causing trigger number 2 to fire. We can then configure trigger number 2 to have an action that releases the correct timeline. It is sometimes easier to write scripts like this when they are going to be sent out to a customer who does not know how to modify them. In this case all the customer needs to know is to modify the start and release timeline actions in the trigger window if they want to change which timeline is run – they do not need to modify the script.

Track motion sensor activity over a period of time

A foyer has 8 pressure pads under the carpet connected to the contact closure inputs of the LPC. We need to count how many times the pressure pads are activated in any 15 second period as a simple measure of activity in the foyer. One of 4 timelines should be selected based on the level of activity.

Hopefully by now you have a pretty good idea of how you could keep a count of the number of digital inputs using script. The new element here is a need for a 15 second timer. We don't do this using script alone but make use of the timeline facilities the LPC already offers.

First the easy bit – for each digital input there is a trigger and they all run the same very simple script:

```
count = count + 1
```

Create a timeline that has no lighting programming but has a flag at 15 seconds. We set the timeline to loop and add a startup trigger that runs it. A trigger on that flag will now fire every 15 seconds while the LPC is running and we set it to run the following script:

```
-- make sure that count has a value (not first time)
if count then
  -- decide which of the 4 timelines to run
  if count < 5 then
    start_timeline(1)
  elseif count < 10 then
    start_timeline(2)
  elseif count < 15 then
    start_timeline(3)
  else
    start_timeline(4)
  end
end

-- now reset count
count = 0
```

Ideally we should also run this script on startup to initialise count – otherwise any digital inputs during the first 15 seconds will try to use count before it has a value and the script will fail. (This will not harm anything or cause the LPC to fail – it will just leave rude messages in the activity log.)

Initially we have got the timer timeline running continuously. What if we only wanted to count for 15 seconds from the first digital input and then stop and wait?

First we would modify the timer timeline (say it is number 5) so that it no longer loops. Then we would modify our script on the digital inputs to be:

```
if count == 0 then
  start_timeline(5)
end

count = count + 1
```

Storing Data to the Memory Card

In the event that our controller reboots, we want it to start running the timeline that was running prior to the reboot.

The Lua library contains functions that make it possible to read and write files to the device the Lua is running on. This includes reading and writing files on the Controller's Memory card.

```
running = 1
```

This variable will be used to store the number of the timeline that was started most recently,using a Timeline Started Trigger (set to Any) with a Run Script action as below:

running = get_trigger_variable(1)

Storing data on the memory card involves two steps, writing to the card and reading back from the card.

```
function writeToCard() -- Write the running timelines table to the
memory card
      file = io.open(get_resource_path("timeline.lua"), "w+") -- Open
or create a file (in write mode) called timeline.lua
      if (file ~= nil) then -- Ensure the file has been opened
            io.output(file) -- Set the open file as the default
output location for the io library
            io.write("running = " .. running) -- Write the line
"running = [value]" to the file. The value will be the value of the
variable
            io.flush() -- Clear the output buffer
            io.close() -- Close the file
      end
end
```

Whenever the function writeToCard is called it will store the current value of the variable running to the memory card in a file called timeline.lua. The file will be in the format:

```
running = [value]
```

This is the syntax for defining a variable in Lua and means that if we run the file on startup, it will set the variable running to be the value stored in the file (with no parsing required)

```
function readFromCard() -- Read the stored running timelines table
and start the timelines specified
      file = io.open(get_resource_path("timeline.lua"),"r") -- Open
the file timeline.lua (in read mode) if it exists
      if file ~= nil then -- Ensure the file has been opened
            dofile(get_resource_path("timeline.lua")) -- Run the file
to set the variable
      end
      get_timeline(running):start() -- Start the timeline stored in
the running variable
end
```

Whenever the function readFromCard is run, it will find the file called timeline.lua, run it so that the stored variable is set on the controller and then start the relevant timeline

Inverting a DMX input before it is used with a Set Intensity action

A client's existing DMX control system is connected to the LPC's DMX input expansion module. Channel 12 controls the houselights and they want the intensity of our LED installation to increase as the houselights go to black out and to decrease as the houselights come on to full.

We can configure a DMX Input trigger so that it fires any time the value of channel 12 changes. As a side effect the DMX Input trigger will always capture the value of the channel it is watching as a trigger variable. If the client wanted the intensity of the LEDs to vary with

the houselights we could put a Set Intensity action directly onto this trigger and take the level from the input. However the client wants an inverse relationship – so instead we will put a simple script in between to invert the value.

The DMX input trigger will run the following script:

```
-- get the value of the DMX channel from variable
chan = variable[1]

-- invert it
chan = 255 - chan

-- now pass it to a second trigger
inject_trigger(2,chan)
```

Trigger 2 can be a soft trigger, but its action should be Set Intensity taking a level from the input. The inject_trigger function allows us to pass additional arguments (after the trigger number) that will be available to the action as trigger variables.

Interpreting data from a wind direction sensor

An LPC is controlling an LED façade on all four sides of a tower block. A wind direction sensor is connected to the LPC via RS232. Every second it sends the character 'X' then a 3 digit number to the LPC which is the wind direction in degrees. The client wants the windward side of the building to always be red, the leeward side to be blue and the other two sides to be green.

We'll create four timelines that correspond to the correct lighting for the wind on each face of the building. We then need an RS232 input trigger that matches the three digit decimal number as a wildcard so that it will be stored as a trigger variable. (The wildcard for this would be "X<3d>" – for more information on this refer to the manual.) Whenever the RS232 input trigger makes a successful match it will run the following script:

```
-- get the value in degrees
dir = variable[1]

-- which face is getting the wind?
if dir < 90 then
  get_timeline(1):start()
elseif dir < 180 then
  get_timeline(2):start()
elseif dir < 270 then
  get_timeline(3):start()
else
  get_timeline(4):start()
end
```

Of course here I have assumed that the corners of the building neatly line up with 0°,90°,180° and 270°. What if they are at 45°,135°,225° and 315°?

```
-- get the value in degrees
dir = variable[1]

-- which face is getting the wind?
if dir < 45 or dir >= 315 then
  get_timeline(1):start()
elseif dir < 135 then
```

```
  get_timeline(2):start()
elseif dir < 225 then
  get_timeline(3):start()
else
  get_timeline(4):start()
end
```

As the wind direction sensor is sending us data every second, when the wind direction is very near a corner then the lights are changing back and forth a lot. The client doesn't like this and wants it limited so that it will only change at most every minute. How should we do that?

Lots of ways to solve this, but one option using script would be:

```
-- get the current minutes
now = time.get_current_time().minute

if now == lastMinute then
  return          -- this will exit the script early
end

-- store in lastMinute, then continue as before
lastMinute = now

-- get the value in degrees
dir = variable[1]

-- which face is getting the wind?
if dir < 45 or dir >= 315 then
  get_timeline(1):start()
elseif dir < 135 then
  get_timeline(2):start()
elseif dir < 225 then
  get_timeline(3):start()
else
  get_timeline(4):start()
end
```

But actually a better solution in this case would probably use a dummy timeline that runs for 1 minute (place a flag at 1 minute to force its length). Add a "timeline is running" condition to the RS232 input trigger that only allows the trigger to fire if the dummy timeline is *not* running. And add a second action to the trigger that starts the dummy timeline. This will ensure that the script is only allowed to run once every minute.

This is probably the better solution because it avoids having a script running every second for no reason. There is some overhead involved in running scripts and it is best to keep the number of scripts that run to the minimum necessary – particularly if the LPC is heavily loaded doing lighting effects at the same time.

Using a table of times for high and low tide

An LPC is controlling the lighting on a bay bridge. The client wants a lighting effect to run at high tide and another effect at low tide. The client has provided tide tables for the entire year and plans to update the tables each year.

A lot of the fun in this situation is in converting the data from the format in which it is provided into a Lua table. This is usually an exercise in Excel and search and replace tools, which I won't cover here. So let's assume we have generated a Lua file of the form:

```
high_tides = {
15,11,08,13,45,   -- 13:45, 8 November 2015
15,11,09,00,36,   -- 00:36, 9 November 2015
15,11,09,11,21,   -- 11:21, 9 November 2015
etc…
}
```

It's best to put large tables like this in their own separate files and run them as separate scripts on startup. Loading a large table like this into memory will take a noticeable amount of time, so you certainly don't want to do that more than once. Also if the tide tables are going to be changed each year then keeping them in a separate file minimises what has to be changed.

We would then also have a realtime trigger that fires every minute and runs the script:

```
-- check if our index variable is initialised
if not h then
  h = 0
end

-- make sure we don't run off the end of the table
while high_tides[h*5] do
  -- compare the current time against table
  year = time.get_current_time().year
  if high_tides[h*5] > year then
    -- not yet reached this entry
    return
  elseif high_tides[h*5] == year then
    month = time.get_current_time().month
    if high_tides[(h*5)+1] > month then
      -- not yet reached this entry
      return
    elseif high_tides[(h*5)+1] == month then
      day = time.get_current_time().monthday
      if high_tides[(h*5)+2] > day then
        -- not yet reached this entry
        return
      elseif high_tides[(h*5)+2] == day then
        hour = time.get_current_time().hour
        if high_tides[(h*5)+3] > hour then
          -- not yet reached this entry
          return
        elseif high_tides[(h*5)+3] == hour then
          minute = time.get_current_time().minute
          if high_tides[(h*5)+4] > minute then
            -- not yet reached this entry
            return
          elseif high_tides[(h*5)+4] == minute then
            -- found a match
            get_timeline(1):start()
              h = h+1
              return
```

```
            end
          end
        end
      end
    end
    -- not yet reached this entry
    h = h+1
  end
```

This may look rather dense and scary – but on closer inspection you'll see that it is just a lot of if tests nested inside each other to check each part of the date and time against the data table. When we detect that the current time is later than the current entry in the table then we move on to looking at the next entry.

Implementing an interactive game for a Science Museum

In an exhibit children are posed questions and have to select answers from an array of numbered buttons. The buttons are large with RGB backlights that are controlled by an LPC to highlight choices and indicate right and wrong answers. Questions are displayed by a slide projector which is under RS232 control from the LPC. The buttons are wired to contact closures on the LPC and on RIOs, so that the LPC can check answers and determine the progress of the game accordingly. The lighting in the rest of the room is designed to mimic a popular TV quiz show to retain the children's interest, with different timelines for each stage of the game.

I am not going to work through this example – but the key point is that it should now be clear to you that an LPC could be used to implement this sort of advanced interactive exhibit with the use of script. Try breaking down the problem into discrete parts and you will find that no individual part of this is difficult – although getting it all to function together reliably would no doubt require a lot of work. The LPC is a viable alternative to custom software running on a PC and has clear advantages in terms of durability and cost.

More information

In this document we have only covered the basic concepts that are needed to understand or write useful scripts for the LPC. For more extensive information on the Lua language there are two documents, both of which are available online at http://www.lua.org or can be bought as books from Amazon.

Lua 5.3 Reference Manual

Programming in Lua

Note that Pharos currently uses Lua 5.3

Full details of the specific Pharos functions are in the reference section of the software manual, available via the Main menu in Designer or at http://www.pharoscontrols.com.

Finally, our support team are happy to advise on scripting problems or to look over your scripts. Please email them to us at support@pharoscontrols.com.