1 DESCRIPTION

The AudioScience Lua scripting extensions allows an AudioScience network interface to automatically perform a user-specified set of actions when a particular condition occurs. For example, an opto input triggered from a push button can cause one or more relays on the network interface to close. The same event could also cause one or more relays on a remote node to close.

2 FEATURES

- Runs on AudioScience network interfaces including ASI2416 (Hono Custom) and Hono Mini
- Uses Lua scripting language to specify inputs and actions
- Actions propagate across the network; i.e. an opto change on one node can cause a relay to activate on any node on the network
- Lua script source and compiled code reside on the network interface
- Script editing and testing supported in ASIControl

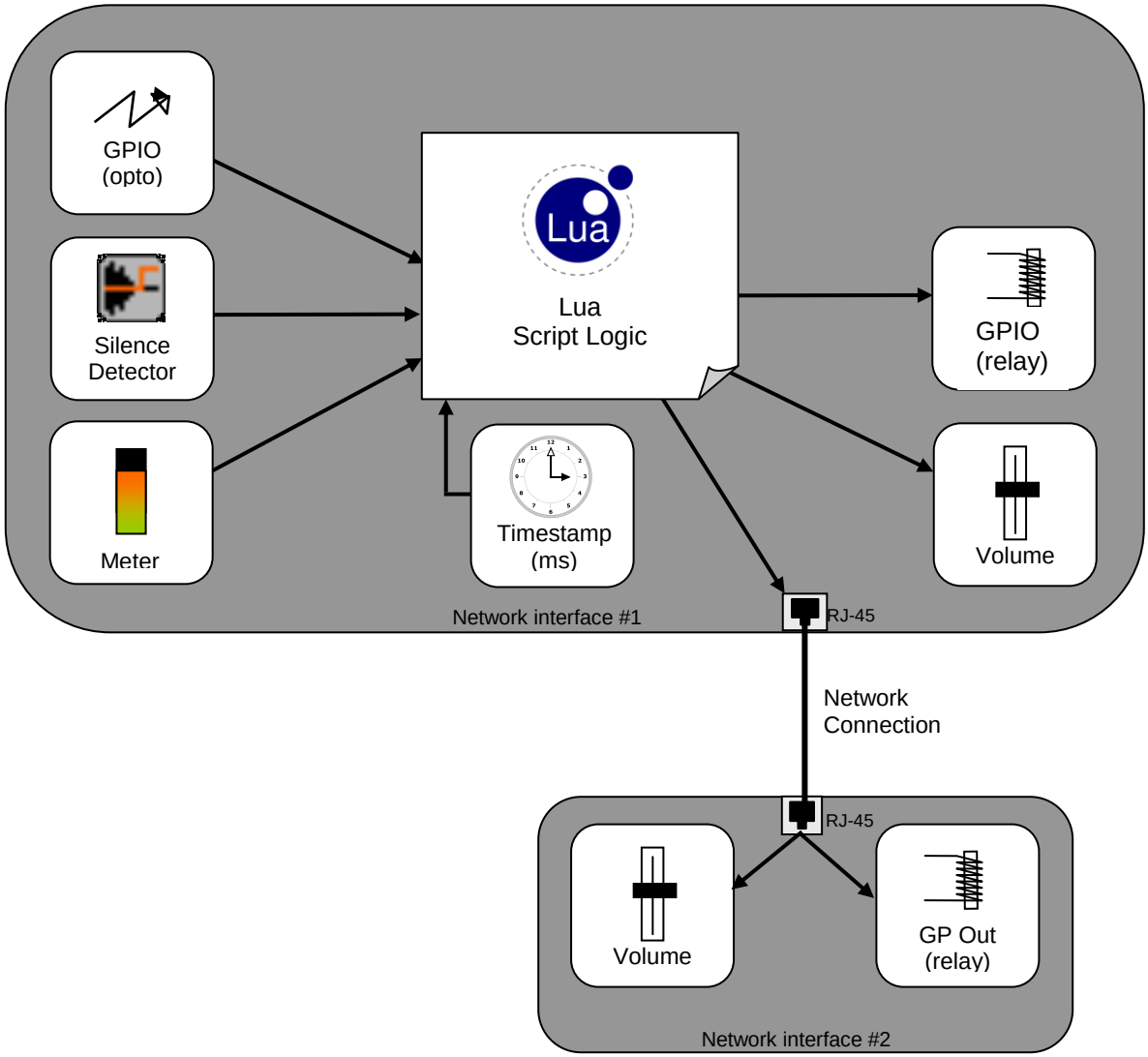


Figure 1. Inputs and outputs to and from a lua script on an AudioScience network interface

3 REVISIONS

Date	Description
22 November 2011	4.10 firmware release using Lua v1.0 interface
March 2016	4.18 firmware release using Lua v2.0 interface
11 January 2017	Added notes re: size limits of scripts

4 CONTENTS

5 AUDIOSCIENCE LUA IMPLEMENTATION

Starting with driver release 4.10.00, Lua scripting is supported on many AudioScience networked audio interfaces. Driver release 4.18.00 supports v2 of the AudioScience Lua runtime. Both versions are covered in this document. In general changes are minimal. The motivation for v2 was improved error handling support when a networked device becomes unavailable for some reason.

The AudioScience Lua implementation supports the standard Lua environment and is based on Lua version 5.1.4 (see References section). However, the following libraries are not available:

- mathematical functions (sin, log, etc.)
- package library
- string manipulation
- table manipulation
- input and output
- operating system facilities
- debug facilities

Note1: print( ) is supported when running a script in ASIControl. On the AudioScience network node the output from the print( ) command is displayed in the Messages section of the Lua block in ASIControl (see 6.3.7).

Note2: All lua indices start at “1” so the first opto would be referred to as opto[1].

5.1 Differences between v1 and v2

- asi.new() now returns and error if the control lookup fails. Additionally the error will not cause the Lua script to exit. V1 script source will still work ok.
- asi.get() now returns an error. This can only occur if the script is being tested in ASIControl.
- asi.events() now returns an error code.
- locahost can be used to refer to the local adapter’s IP address

Script source code is 100% backwards compatible because the additional returned parameters will be ignored.

5.2 Language Extensions

AudioScience’s lua implementation adds a library called “asi” to the runtime environment. Since the library is loaded by AudioScience’s runtime environment, there is no need to use the lua module command to load the asi library.

5.2.1 Error Handling

There are several classes of errors that can happen in a script.

Syntax error. Will be caught when the script is compiled.

Non-fatal runtime error (v2 only). Generally these are the result of loss of network connection to a remote adapter. Methods that normally return nothing return an error code. Methods that normally return some value(s) return nil, error_code if the error invalidates the normal return value.

Error codes:
109:
110: Network timeout, connection lost to remote adapter?
900: No adapter found at specified address

Runtime exception. Typically the result of a malformed URI, e.g. referring to a non-existent control.

If one of these occurs the script will stop running.

5.2.2 URIs

All available controls are referenced using the "hpi://" URI scheme

"hpi://" + "ip_address" + "control_path"

For example "hpi://123.4.5.6/adapter/gpio/inputs

Only a subset of adapter controls are available to Lua scripting.

Note that “localhost” is a supported IP address in Lua runtime V2.

Valid control paths
('#' denotes the 1-based index of the control, i.e. 1, 2, ... etc)

GPIOs:

/adapter/
  gpio/
    inputs
    outputs

Inputs:

/aesebu_in/#/
/analog_in/#/
/cobranet_in/#/
/internal_in/#/
/microphone/#/
  meter/peak
  meter/rms
  volume/gain
  volume/mute

Outputs:

/aesebu_out/#/
/analog_out/#/
/cobranet_out/#/
  meter/peak
  meter/rms
  silence_detector/state
  silence_detector/enable
  volume/gain
  volume/mute

Matrix mixer volumes:

/[Input]/#/[Output]#/
  volume/gain
  volume/mute

E.g. /cobranet_in/1/analog_out/2/volume/gain

5.2.3 Method asi.version(v) (V2 only)

Retuns the version of the Lua runtime that is being used. There is no requirement to call this function for correct operation, but if you wish to be sure your script is running on a compatible runtime, it can be useful.

5.2.3.1 Parameters

- version: the runtime version this script is expected to run on

5.2.3.2 Return

- The runtime interpreter version in use

5.2.3.3 Example

```
script_api_version = 2
interpreter_version = asi.version(script_api_version)
if script_api_version ~= interpreter_version then
   print("API version mismatch")
end
```

5.2.4 Method asi.new(uri)

Creates a handle to the specified object.

5.2.4.1 Parameters

A URI string specifying the object to use.

5.2.4.2 Return

- handle (v1 and v2): A lua object of type metatable. Optionally, a lua table of metatables is returned if the URI specifies an indexed object (like GP In optos).
- nil, error (v2): a non fatal error occurred.
- <exception>: The uri is malformed or does not exist on the target adapter.

5.2.4.3 Example

```
Version 1 code
local_ip = "192.168.1.53"
opto = asi.new("hpi://"..local_ip.."/adapter/gpio/inputs")
silence_detector =
      asi.new("hpi://"..local_ip.."/analog_out/1/silence_detector/state")

Version 2 code
opto, err_opto = asi.new("hpi://localhost/ adapter/gpio/inputs")
silence_detector, err_silence =
      asi.new("hpi://localhost/analog_out/1/silence_detector/state")
```

In the v2 example above the err_XXXX returns can be used to retry to the asi.new() operation at a later time. The err_XXXX return also indicates that the returned handle should not be used.

5.2.5 Method asi.get()

Reads the current setting of the object. Reading control values (e.g. meter, gpio) is only supported on the local adapter. This is so that polling for a value change does not generate network traffic. (Also local reads are significantly more efficient).

Note that reading from a remote control may give no error, but will not actually read from the remote adapter.

5.2.5.1 Parameter

An object (metatable) that was created using asi.new() as specified above. Using lua's implied reference to self object, `asi.get(obj)` can be expressed as `obj:get()`.

5.2.5.2 Return

- state, changed: The current state of the object and a Boolean changed flag. The type of the state depends on the object being referenced.
- nil, error_number (v2 only): An error occurred getting the object's state

5.2.5.3 Example

```
V1 & V2
state,changed = opto[1]:get()

V2 (error handling)
state,changed = opto[1]:get()
if state == nil then
      print("Error", changed, "reading opto")
elseif changed then
      print("Changed to", state)
      r_relay[1]:set(state)
else
      --print("No input change")
end
```

5.2.6 Method asi.set()

Sets the value of specified object.

5.2.6.1 Parameters

An object (metatable) that was created using asi.new() as specified above and the value to set. Using lua's implied reference to self object, `asi.set(obj,value)` can be expressed as `obj:set(value)`.

5.2.6.2 Return

- error_code : 0 = no error, non-zero, the set operation has failed.

5.2.6.3 Example

```
V1
relay[1]:set(state)
V2
err = relay[1]:set(state)
```

5.2.7 Method asi.events()

Returns true while there are events to process. This is used to bracket an infinite while loop that reads inputs, performs logical operations, and sets outputs. In the background the call to asi,events() sleeps for 100 milliseconds and updates the value that is returned by subsequent calls to asi.time().

Any potentially infinite loop MUST call this function and exit if the return value is false.

5.2.7.1 Parameter

None.

5.2.7.2 Return V1

True or false.

5.2.7.3 Return V2

- keep_going, err: The keep_going_flag is true if script can keep running, false if script must exit.

The error code pertains to updates to remote relays. If it is non-zero, the remote relays may not have been updated.

5.2.7.4 Example v1

```
while ( asi.events() ) do
-- read inputs
-- perform logic
end
```

5.2.7.5 Example v2

```
while true do
  keep_going, err = asi.events()
  if not keep_going then
    break
  end
-- read inputs
-- perform logic
```

- end

5.2.8 Method asi.time()

Reads the current time in milliseconds. This returns a 32-bit number (counter) that wraps around, so care must be taken when doing operations on the returned time value. See `asi.difftime()` and `asi.addtime()` methods. Although the timer has a resolution of a millisecond, lua scripts are not run every millisecond. Greater than a duration comparisons should therefore be used, not equal to comparisons.

5.2.8.1 Parameter

None.

5.2.8.2 Return

Time stamp in millisecond ticks.

5.2.8.3 Example

```
current_time = asi.time()
```

5.2.9 Method asi.difftime()

Takes the difference between time objects that both represent a timestamp in milliseconds.

5.2.9.1 Parameters

Two timestamps, A and B.

5.2.9.2 Return

The time difference in milliseconds. If input parameters are called A and B, then the returned time difference is A – B.

5.2.9.3 Example

```
time_reference = asi.time()

while ( asi.events() ) do

      current_time = asi.time()

      -- have 3 seconds passed ?
      if asi.difftime( current_time, time_reference) > 3000 then
            -- take action(s)
            -- update the time reference for next period
            time_reference = asi.addtime(time_reference,3000)
      end
end
```

5.2.10 Method asi.addtime()

Add two millisecond timestamps together and return the result.

5.2.10.1 Parameter

Two timestamps, A and B.

5.2.10.2 Return

The sum of 2 timestamps in milliseconds. If parameters are A and B, returned time sum is A + B. The return timestamp may be less than both A and B if the sum causes the 32-bit unsigned timestamp presentation to wrap.

5.2.10.3 Example

```
time_reference = asi.time()
while ( asi.events() ) do

      current_time = asi.time()

      -- have 3 seconds passed ?
      if asi.difftime( current_time, time_reference) > 3000 then
            -- take action(s)
            -- update the time reference for next period
            time_reference = asi.addtime(time_reference,3000)
      end
end
```

5.3 Script Layout

Language extensions outlined in the previous section shape the structure of lua script running on an AudioScience network node. Scripts are broken down into an opening section followed by a looping section. Be sure to observe size limits as noted in section “Using ASIControl to Write Scripts” below.

5.3.1 Opening section

The opening section uses the `asi.new()` call to map AudioScience control objects to lua objects that can be manipulated later on the script. All calls to create lua objects should be completed before the input processing section.

Any required initial operating conditions can also be set before the following looping section is entered. For example, all input microphones can be muted.

5.3.2 Looping section

The looping section is where the processing occurs. Inputs are read and actions taken according to the rules and conditions written into the lua script. Looping continues until the script is halted.

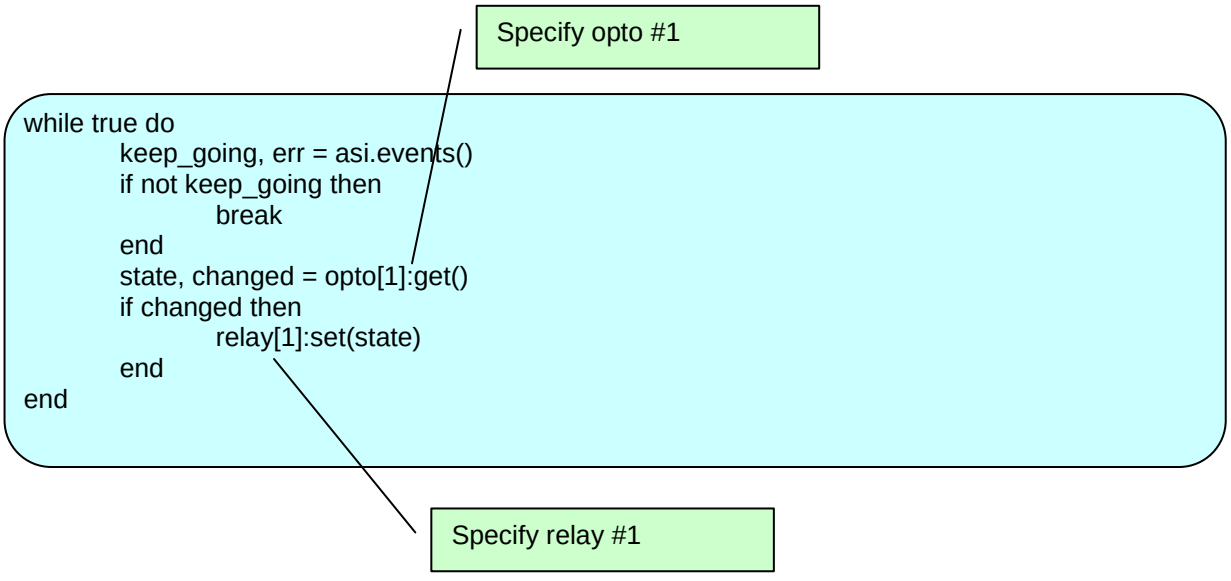
5.4 Examples (Lua AudioScience runtime v2)

5.4.1 Single opto input to relay output – v2

Open the opto and relay handles.

```
opto, oerr = asi.new("hpi://"..ip.."/localhost/gpio/inputs")
relay, rerr = asi.new("hpi://"..ip.."/localhost/gpio/outputs")
```

Create the processing loop.



Final code:

```
-- example, single opto input to relay output
opto = asi.new("hpi://localhost/adapter/gpio/inputs")
relay = asi.new("hpi://localhost/adapter/gpio/outputs")
while true do
        keep_going, err = asi.events()
        if not keep_going then
                break
        end
        state, changed = opto[1]:get()
        if changed then
                relay[1]:set(state)
        end
end
```

5.4.2 Single opto input to multiple relay outputs – v2

This example uses the same object handles as the previous example. The action part of the processing loop is altered to set relay outputs 1-4 instead of just relay output number 1.

Final code:

```
-- example, single opto input to multiple relay outputs
opto = asi.new("hpi://localhost/adapter/gpio/inputs")
relay = asi.new("hpi://localhost/adapter/gpio/outputs")
while true do
        keep_going, err = asi.events()
        if not keep_going then
                break
        end
        state, changed = opto[1]:get()
        if changed then
                relay[1]:set(state)
                relay[2]:set(state)
                relay[3]:set(state)
                relay[4]:set(state)
        end
end
```

5.4.3 Single opto input with pulse extended relay output – v2

Pulse extending an opto input requires the use of timing information within the lua script. The current time stamp is queried using:

```
current_time = asi.time()
```

When the opto is triggered we record the current time along with the fact that we are now pulse extending using the following code snippet:

```
time_reference = current_time
pulse_extending = true
```

Future iterations of the script loop examine the elapsed time and take appropriate actions using the following code (assuming a 10 second pulse extension):

```
if asi.difftime( current_time, time_reference ) > 10000 then
        pulse_extending = false
        relay[1]:set(false)
end
```

Final code:

```
-- example, single opto input with 10 second pulse extended relay output
opto = asi.new("hpi://localhost/adapter/gpio/inputs")
relay = asi.new("hpi://localhost/adapter/gpio/outputs")
pulse_extending = false
while true do
        keep_going, err = asi.events()
        if not keep_going then
                break
        end
        current_time = asi.time()
        state, changed = opto[1]:get()
        if changed then
                if state then
                        time_reference = current_time
                        pulse_extending = true
                        relay[1]:set(true)
                end
        else
                if pulse_extending then
                        if asi.difftime( current_time, time_reference ) > 10000 then
                                pulse_extending = false
                                relay[1]:set(false)
                        end
                end
        end
end
```

5.4.4 Single opto input to relay output on remote device – v2

Interfacing to a remote device requires only an IP address change in the string passed in to asi.new(). For this example, the IP address of the remote device is defined as:

```
ip_other = "192.168.1.70"
```

An object handle to the GP Out on the remote device is obtained by going:

```
relay, err = asi.new("hpi://"..ip_other.."/adapter/gpio/outputs")
```

Putting it all together, final code becomes:

```
-- example, single opto input to single relay output on remote device
ip_other = "192.168.1.70"
opto = asi.new("hpi://localhost/adapter/gpio/inputs")
relay, rerr = asi.new("hpi://"..ip_other.."/adapter/gpio/outputs")
while true do
        keep_going, err = asi.events()
        if not keep_going then
                break
        end

        state, changed = opto[1]:get()

        -- keep trying to open relay until success
        if not relay then
                relay, rerr = asi.new("hpi://"..ip_other.."/adapter/gpio/outputs")
        else
                if changed then
                        relay[1]:set(state)
                end
        end
end
```

5.4.5 Silence detector to relay output – v2

A silence detector returns a true or false state, exactly the way an opto does. A handle to the silence detector state for analog output number 1 is obtained using:

```
silence_detect = asi.new("hpi://localhost/analog_out/1/silence_detector/state")
relay = asi.new("hpi://localhost/adapter/gpio/outputs")
```

Note, ASIControl should be used to enable the silence detector on as it is disabled by default. The control loop becomes:

```
while true do
        keep_going, err = asi.events()
        if not keep_going then
                break
        end     state, changed = silence_detect:get()
        if changed then
                relay[1]:set(state)
        end
end
```

5.4.6 Push to talk microphone – v2

Read an opto and when the opto is trigger unmute the microphone.

```
opto = asi.new("hpi://localhost/adapter/gpio/inputs")
mute = asi.new("hpi://localhost/analog_in/1/volume/mute")
mute:set(false)
```

Note, ASIControl should be used to enable the silence detector on as it is disabled by default.

```
while true do
        keep_going, err = asi.events()
        if not keep_going then
                break
        end
        state, changed = opto[1]:get()
        if changed then
                mute:set(state)
        end
```

5.4.7 Voice activated microphone(s) – v2

When the input meter level measured in RMS exceeds a specified level, un-mute the microphone.

```
meter = asi.new("hpi://localhost/analog_in/1/meter/rms")
mute = asi.new("hpi://localhost/analog_in/1/volume/mute")
db_FS_mute_threshold = -20
```

Processing loop.

```
while true do
        keep_going, err = asi.events()
        if not keep_going then
                break
        end
        rms = meter:get()
        mute:set( rms < db_FS_mute_threshold )
end
```

5.5 Examples (Lua AudioScience runtime v1)

5.5.1 Single opto input to relay output – v1

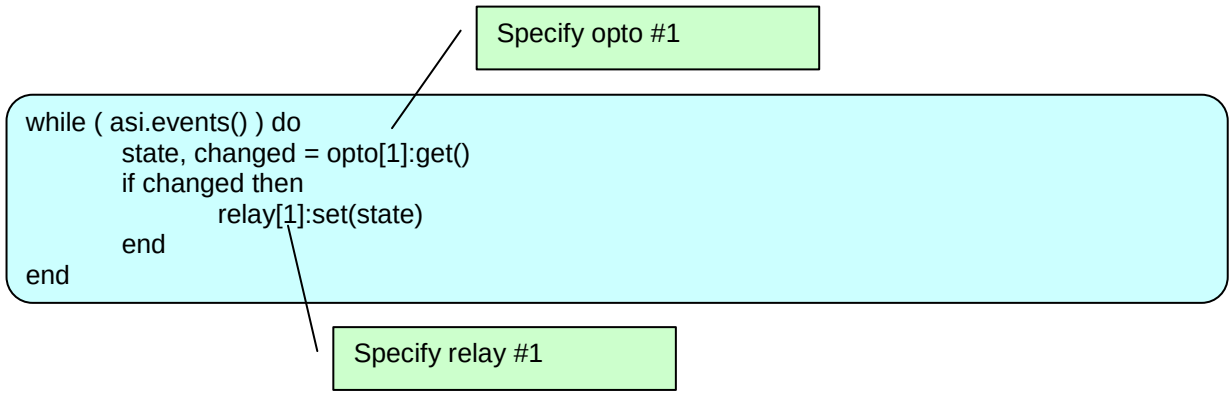
Define the IP address of the ASI2416 that will run the script.

```
ip = “192.168.1.53”
```

Open the opto and relay handles.

```
opto = asi.new("hpi://"..ip.."/adapter/gpio/inputs")
relay = asi.new("hpi://"..ip.."/adapter/gpio/outputs")
```

Create the processing loop.



Final code:

```
-- example, single opto input to relay output
local_ip = "192.168.1.53"
opto = asi.new("hpi://"..ip.."/adapter/gpio/inputs")
relay = asi.new("hpi://"..ip.."/adapter/gpio/outputs")
while ( asi.events() ) do
        state, changed = opto[1]:get()
        if changed then
                relay[1]:set(state)
        end
end
```

5.5.2 Single opto input to multiple relay outputs – v1

This example uses the same object handles as the previous example. The action part of the processing loop is altered to set relay outputs 1-4 instead of just relay output number 1.

Final code:

```
-- example, single opto input to multiple relay outputs
ip = "192.168.1.53"
opto = asi.new("hpi://"..ip.."/adapter/gpio/inputs")
relay = asi.new("hpi://"..ip.."/adapter/gpio/outputs")
while ( asi.events() ) do
        state, changed = opto[1]:get()
        if changed then
                relay[1]:set(state)
                relay[2]:set(state)
                relay[3]:set(state)
                relay[4]:set(state)
        end
end
```

5.5.3 Single opto input with pulse extended relay output – v1

Pulse extending an opto input requires the use of timing information within the lua script. The current time stamp is queried using:

```
current_time = asi.time()
```

When the opto is triggered we record the current time along with the fact that we are now pulse extending using the following code snippet:

```
time_reference = current_time
pulse_extending = true
```

Future iterations of the script loop examine the elapsed time and take appropriate actions using the following code (assuming a 10 second pulse extension):

```
if asi.difftime( current_time, time_reference ) > 10000 then
        pulse_extending = false
        relay[1]:set(false)
end
```

Final code:

```
-- example, single opto input with 10 second pulse extended relay output
ip = "192.168.1.53"
opto = asi.new("hpi://"..ip.."/adapter/gpio/inputs")
relay = asi.new("hpi://"..ip.."/adapter/gpio/outputs")
pulse_extending = false
while ( asi.events() ) do
        current_time = asi.time()
        state, changed = opto[1]:get()
        if changed then
                if state then
                        time_reference = current_time
                        pulse_extending = true
                        relay[1]:set(true)
                end
        else
                if pulse_extending then
                        if asi.difftime( current_time, time_reference ) > 10000 then
                                pulse_extending = false
                                relay[1]:set(false)
                        end
                end
        end
end
```

5.5.4 Single opto input to relay output on remote device – v1

Interfacing to a remote device requires only an IP address change in the string passed in to asi.new(). For this example, the IP address of the remote device is defined as:

```
ip_other = "192.168.1.70"
```

An object handle to the GP Out on the remote device is obtained by going:

```
relay = asi.new("hpi://"..ip_other.."/adapter/gpio/outputs")
```

Putting it all together, final code becomes:

```
-- example, single opto input to single relay output on remote
device
ip = "192.168.1.53"
ip_other = "192.168.1.70"
opto = asi.new("hpi://"..ip.."/adapter/gpio/inputs")
relay = asi.new("hpi://"..ip_other.."/adapter/gpio/outputs")
while ( asi.events() ) do
        state, changed = opto[1]:get()
        if changed then
                relay[1]:set(state)
        end
end
```

5.5.5 Silence detector to relay output – v1

A silence detector returns a true or false state, exactly the way an opto does. A handle to the silence detector state for analog output number 1 is obtained using:

```
silence_detect = asi.new("hpi://"..ip.."/analog_out/1/silence_detector/state")
relay = asi.new("hpi://"..ip.."/adapter/gpio/outputs")
```

Note, ASIControl should be used to enable the silence detector on as it is disabled by default. The control loop becomes:

```
while ( asi.events() ) do
        state, changed = silence_detect:get()
        if changed then
                relay[1]:set(state)
        end
end
```

5.5.6 Push to talk microphone – v1

Read an opto and when the opto is trigger unmute the microphone.

```
opto = asi.new("hpi://"..ip.."/adapter/gpio/inputs")
mute = asi.new("hpi://"..ip.."/analog_in/1/volume/mute")
mute:set(false)
```

Note, ASIControl should be used to enable the silence detector on as it is disabled by default.

```
while ( asi.events() ) do
        state, changed = opto[1]:get()
        if changed then
                mute:set(state)
        end
end
```

5.5.7 Voice activated microphone(s) – v1

When the input meter level measured in RMS exceeds a specified level, un-mute the microphone.

```
meter = asi.new("hpi://"..ip.."/analog_in/1/meter/rms")
mute = asi.new("hpi://"..ip.."/analog_in/1/volume/mute")
db_FS_mute_threshold = -20
```

Processing loop.

```
while ( asi.events() ) do
        rms = meter:get()
        mute:set( rms < db_FS_mute_threshold )
end
```

5.6 Using ASIControl to Write Scripts

Notes

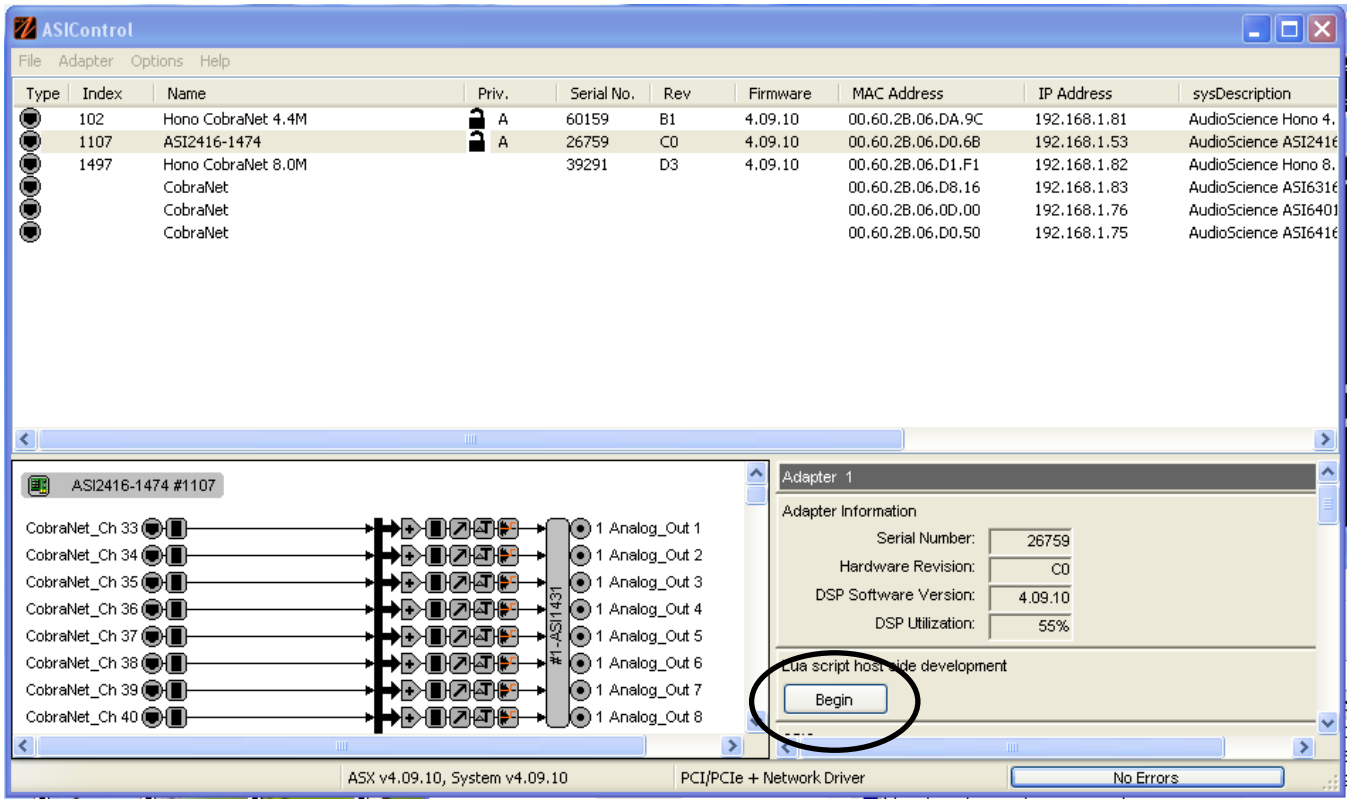
1. Static IP addresses are required for reliable lua script operation! See the device's datasheet for instructions.
2. Lua source code stored on the device has a maximum size limit of 16KB.
3. Lua compiled bytecode has a maximum size limit of 16KB.

ASIControl embeds a text editor that can be used to create and edit lua scripts. The typical development sequence proceeds as follows.

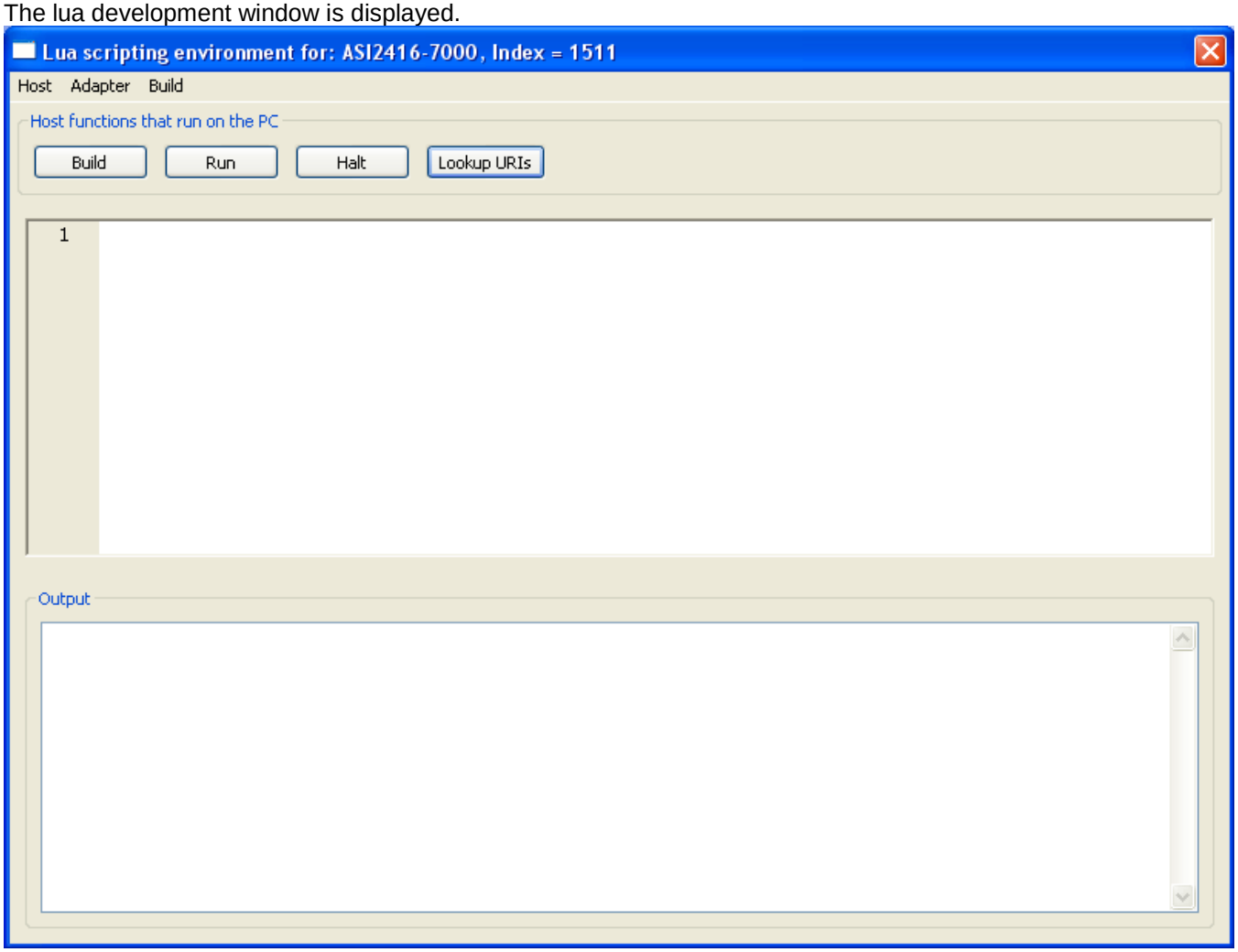
1. Design on paper what inputs should control what outputs for the entire system.
2. Design on paper what inputs should control what outputs for each device.
3. Create the .lua scripts for each device and run them on the host for testing purposes before downloading and running them on the network node.
4. Optionally, save the .lua source code on the network node.

5.6.1 Opening the script editor

Open the lua development environment by clicking on the Begin button as shown in the below figure.



The lua development window is displayed.



Starting from the top left we have a “File” menu for loading and saving lua source files to and from the host PC hard-drive. Next we have the “Adapter” menu which operates the device’s lua file storage. It supports saving lua source files and lua byte code to the network interface.

The “Host functions…” section controls host operations for script development. Operations are :

- Compile - converts lua source code to byte code and produces a .luac output file with the same name as the .lua script file. A compile operation must be performed before the script can be run.
- Run – starts a compiled lua script running on the host.
- Halt – stops a lua script running on the host. Since the structure of all 2416 lua scripts contains a repeating loop so that they keep running, a “Halt” operation is always required to terminate script execution.
- Lookup URIs – supports looking up uri strings for the currently attached network interfaces.

The upper of the 2 large windows above is the text editor, while the lower is output from the engine that runs the lua script.

5.6.2 Opening an existing lua source file from disk

Choose Host → File Open to open a lua source file from the local host PC's hard drive. lua source files are assumed to use the extension .lua.

5.6.3 Saving to disk

Choose Host → File Save to save the lua source file being edited to the local hard drive. Note that compiling the source will automatically save the source to the hard drive.

5.6.4 Saving script to AudioScience audio network node

Choose Adapter → Save source to save the local lua script to the remote network node

Note: For an ASI2300 family audio network node, the maximum supported file size is 16 kBytes

Note that there is no requirement to save the lua source to the network node for lua scripting to work correctly. Only the bytecode need be saved.

5.6.5 Reading script to AudioScience audio network node

Choose Adapter → Load source to retrieve lua script source from the remote node.

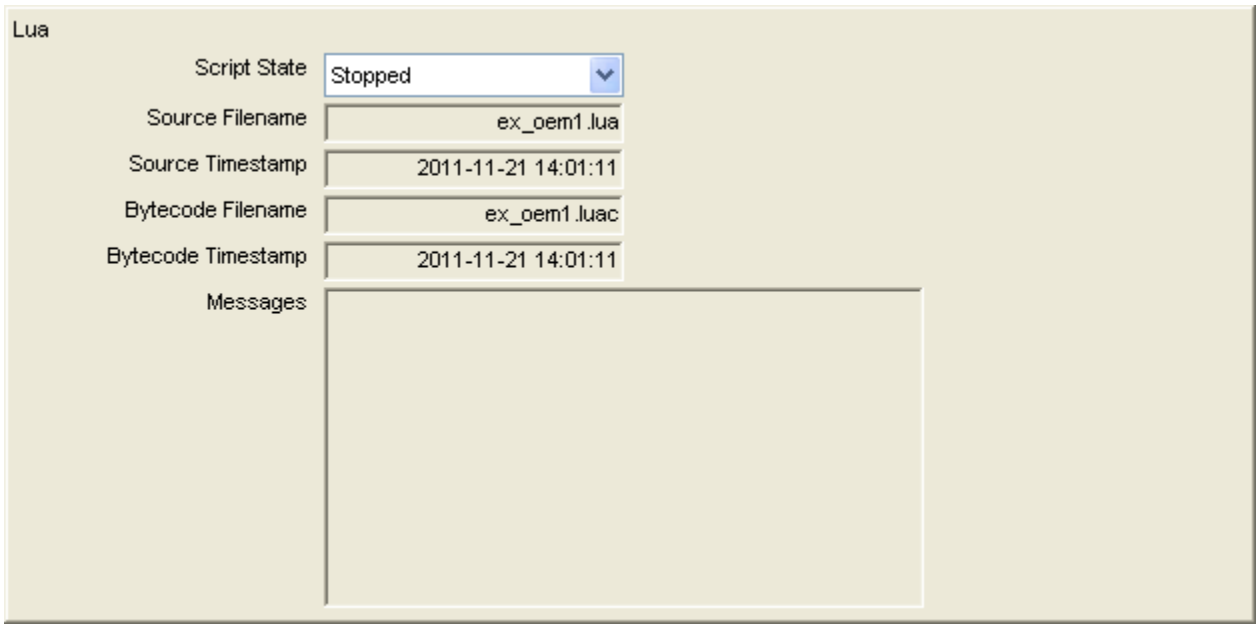
5.6.6 Running the script on the PC for testing purposes

The steps for developing a lua script on an AudioScience network interface are:

1. Start with a lua example that is close to what you are trying to implement. Load it into the editor.
2. Save it under a new name using the "Save As" option.
3. Take care to adjust IP address(es) .
4. Make functional changes as required. Use the "Lookup URIs" option as required to obtain object uris.
5. Compile the script.
6. Run the script. If there are syntax errors go back to step 4.
7. Try to manipulate inputs so that you can verify the script is working.
8. If all is working press "Halt".
9. Save the script to the network node.

5.6.7 Running the script on the network interface

ASIControl shows a “lua” block on network interfaces that support lua scripting. The names of lua source files and lua byte code files that are stored on the network interface are displayed:



Most lua control block functions are obvious. The Script Status option is used to control the script status and the Messages section shows message from the lua runtime engine running on the network node. If a script is selected to be run, the setting will be preserved through a power down cycle of the network interface.

5.7 Lua License

5.8 References

For more information on lua, see http://www.lua.org/.

<end>