

openLuup

@akboer, May 2020

Contents

| | |
|-------------------------------------------------------------------------|-----------|
| openLuup | 1 |
| Contents | 1 |
| Overview | 3 |
| 2020, Release x.xx (v20xx.xx) | 4 |
| Installation | 5 |
| 1. Lua installation | 5 |
| 2. openLuup Installation | 6 |
| 3. Making openLuup run at system boot time | 6 |
| 4. Linking to Vera | 7 |
| Backing up the system | 8 |
| More about VeraBridge | 9 |
| 1. Bridging to multiple Veras | 10 |
| 2. Mirroring individual device variables from openLuup to Vera | 10 |
| 3. Synchronising House Mode | 10 |
| Other configuration variables | 11 |
| The Data Historian | 12 |
| The user_data.json file | 14 |
| More about the port 3480 HTTP server | 15 |
| 1. File Access | 15 |
| 2. index.html Files | 15 |
| 3. CGI processing | 16 |
| Something about openLuup log files | 17 |
| More about Scenes | 18 |
| 1. VeraBridge and Remote Scenes | 18 |
| 2. Porting Vera scenes to openLuup | 19 |
| 3. Scene Finalisers – Running Lua Code after Individual Scene Execution | 20 |
| 4. Scene Prolog and Epilog Lua code – Global Functions | 21 |
| Appendix: Starting openLuup at system boot time | 22 |
| systemctl with /etc/systemd/system/openluup.service | 22 |
| /etc/init.d/openLuup | 23 |
| /etc/rc.local | 24 |

| | |
|-----------------------------------------------------------------|-----------|
| Appendix: Configuring key openLuup parameters at Startup | 26 |
| Appendix: Directory Structure and Additional Files | 27 |
| 1. Vera & openLuup directory structures and ancillary files | 27 |
| 2. /usr/bin/GetNetworkState.sh and /etc/cmh/ui | 28 |
| 3. openLuup_reload.bat for Windows | 28 |
| 4. code for /usr/bin/GetNetworkState.sh | 29 |
| Appendix: openLuup SMTP and POP3 (eMail) servers | 30 |
| SMTP server | 30 |
| Testing the SMTP server | 32 |
| POP3 server | 34 |
| Appendix: Using Cameras with openLuup | 35 |
| l_openLuupCamera1.xml implementation file | 35 |
| Configuration | 35 |
| Appendix: openLuup – Databases and Data Visualisation | 36 |
| InfluxDB | 37 |
| DataYours / Graphite | 37 |
| Appendix: Sending and receiving UDP datagrams | 38 |
| Appendix: openLuup plugin Actions and Variables | 39 |
| House Mode related actions and variables | 39 |
| File Management actions and variables | 39 |
| AltUI-related variables | 40 |
| System Status Variables | 40 |
| Appendix: Undocumented features of Luup | 41 |
| Appendix: Unimplemented features of openLuup | 42 |
| 1. unimplemented luup API calls | 42 |
| 2. unimplemented HTTP requests | 42 |
| Appendix: Differences between openLuup and Vera / MiOS | 43 |
| Features of Vera / MiOS not in openLuup | 43 |
| Features of openLuup not in Vera / MiOS | 43 |

Overview

openLuup is an environment which supports the running of some MiOS (Vera) plugins on generic Unix systems (or, indeed, Windows systems.) Processors such as Raspberry Pi and BeagleBone Black are ideal for running this environment, although it can also run on Apple Mac, Microsoft Windows PCs, anything, in fact, which can run Lua code (most things can.) The intention is to offload processing (cpu and memory use) from a running Vera to a remote machine to increase system reliability.

Running on non-specific hardware means that there is no native support for Z-wave, although plugins to handle Z-wave USB sticks support this (the latest openLuup versions support a ZWay plugin for [Z-Wave.me](http://www.z-wave.me) Razberry boards or UZB slicks.) The full range of MySensors (<http://www.mysensors.org/>) Arduino devices are supported though the Ethernet Bridge plugin available on that site. A plugin, VeraBridge, to provide a bi-directional 'bridge' (monitoring / control, including scenes) to remote MiOS (Vera) systems is provided in the openLuup installation.

openLuup is extremely fast to start (a few seconds before it starts running any created devices startup code) has very low cpu load, and has a very compact memory footprint. Whereas each plugin on a Vera system might take ~4 Mbytes, it's far less than this under openLuup, in fact, the whole system can fit into that sort of space. Since the hardware on which it runs is anyway likely to have much more physical memory than current Vera systems, memory is not really an issue.

For the user interface, but we have, courtesy of @amg0, the most excellent AltUI: Alternate UI to UI7 (see the Vera forum board <https://community.getvera.com/c/plugins-and-plugin-development/alternate-ui-to-ui7/>) An automated way of installing and updating the AltUI environment is built-in to openLuup. There's actually no requirement for any user interface if all that's needed is an environment to run plugins. There is a rudimentary console interface for examine some of the system objects and data,

A built-in VeraBridge action is provided to transfer a complete set of uncompressed device files and icons from any Vera on your network to the openLuup target machine. It also supports liking to Vera scenes and/or importing them to run as native openLuup scenes.

openLuup includes a 'Data Historian' which stores previous device variable values in both in-memory cache and (optionally) on-disk archives. It offers an almost configuration-free approach to storing (and retrieving) variable values, and provides plotting through both a Gefena Data Source API (so that data may be easily visualised if you already have a Grafana installation) or plotted directly by HTTP request (using Google Charts.)

For compatibility with Vera systems, openLuup has been written in Lua version 5.1 and requires the appropriate installation to be loaded on your target machine. Installation is not trivial, but it's not hard either.

Please read this document carefully and then follow the installation steps.

2020, Release x.xx (v20xx.xx)

This release is a comprehensive (but not totally complete) implementation of Luup, and includes a set of features which are generally sufficient to run a large number of third-party plugins.

What openLuup does:

- runs on any Unix machine - I often run it within my development environment on a Mac
- also runs under Windows. However, plugins which spawn commands using the `Lua os.execute()` function may not operate correctly
- runs the AltUI plugin to give a great UI experience
- runs the MySensors Arduino plugin (ethernet connection to gateway only) which is really the main goal - to have a Vera-like machine built entirely from third-party bits (open source)
- runs the ZWay plugin to provide basic control of Wave devices through the Z-Wave.me Raspberry board or UZB stick
- includes a bridge app to link to remote Veras (which can be running UI5 or UI7 and require no additional software.)
- runs many plugins unmodified – particularly those which just create virtual devices (eg. DataYours, Netatmo, ...)
- provides access to the AltAppStore where other plugins tested under openLuup by their authors are available for download
- uses a tiny amount of memory and boots up very quickly (a few seconds)
- supports scenes with timers and AltUI-style variable watch triggers
- has its own port 3480 HTTP server supporting multiple asynchronous client requests
- has a fairly complete implementation of the Luup API and the HTTP requests
- has a simple log structure - most events generate just one entry each in the log.
- writes variables to a separate log file for AltUI to display variable and scene changes.
- special camera implementation file with motion sensor triggered by camera
- SMTP/POP3 email server to receive and browse local emails within your LAN
- File retention policy manager with SendToTrash and EmptyTrash actions

What it doesn't do:

- Some less-used HTML requests are not yet implemented (see Appendix.)
- Doesn't support the `<incoming>` or `<timeout>` action tags in service files, but does support the device-level `<incoming>` tag.
- Doesn't directly support local serial I/O hardware (there are work-arounds: ser2net)
- Won't run encrypted, or licensed, plugins.
- Doesn't use lots of memory.
- Doesn't use lots of cpu.
- Doesn't constantly reload (like Vera often does, for no apparent reason.)
- Doesn't do UPnP (and never will.)

See the relevant appendix for more discussion of Vera / openLuup differences.

If you like openLuup, please consider donating something to [Cancer Research UK](http://www.justgiving.com/DataYours/) at <https://www.justgiving.com/DataYours/>

Installation

There are just a few basic installation steps to set up an openLuup system running the AltUI interface and bridging to a remote Vera:

1. install Lua and some of its packages on your target machine
2. install openLuup itself using the `openLuup_install.lua` file
3. install and run the VeraBridge plugin from the AltUI Plugins page

That's all that's needed!

1. Lua installation

For compatibility with Vera systems, openLuup has been written in Lua version 5.1 and requires the appropriate installation to be loaded on your target machine. You will also need the LuaSocket library, LuaFileSystem, and, for secure (SSL) network connections, the LuaSec library. It's not difficult, but the guide assumes some familiarity with Linux-type systems (although much is also applicable to Windows.)

The installation process is target machine dependent (other machines may require source code compilation of the libraries, but that is beyond the scope of this document) but a few of the simpler cases are:

RASPBIAN (RASPBERRY PI)

(Lua 5.1 is pre-installed)

```
# sudo apt-get update
# sudo apt-get install lua-socket
# sudo apt-get install lua-filessystem
# sudo apt-get install lua-sec
```

DEBIAN (BEAGLEBONE BLACK)

```
# apt-get update
# apt-get install lua5.1
# apt-get install lua-socket
# apt-get install lua-filessystem
# apt-get install lua-sec
```

OPEN-WRT (AS USED BY VERA!)

(Lua 5.1 is pre-installed)

```
# opkg update
# opkg install luasocket
# opkg install luafilesystem
# opkg install luasec
```

2. openLuup Installation

This is *very* straight-forward: create a directory **cmh-ludl/** for the openLuup installation on your machine (in your home directory is a good place, or for compatibility with Vera use **/etc/cmh-ludl/** but be careful with permissions) **cd** to it, and retrieve the file **openLuup_install.lua** from the GitHub repository using:

```
wget https://github.com/akbooper/openLuup/raw/master/Utilities/openLuup_install.lua
```

The link below may also be used to retrieve the file:

https://github.com/akbooper/openLuup/raw/master/Utilities/openLuup_install.lua

Run the file using the command line:

```
# lua5.1 openLuup_install.lua
```

If successful, the script produces console output like this:

```
# lua5.1 install.lua
openLuup_install 2016.06.08 @akbooper
getting latest openLuup version tar file from GitHub...
un-zipping download files...
getting dkjson.lua...
creating required files and folders
initialising...
downloading and installing AltUI...
Tue Nov 8 11:08:32 2016 device 2 'openLuup' requesting reload
openLuup downloaded, installed, and running...
visit http://172.16.42.131:3480 to start using the system
#
```

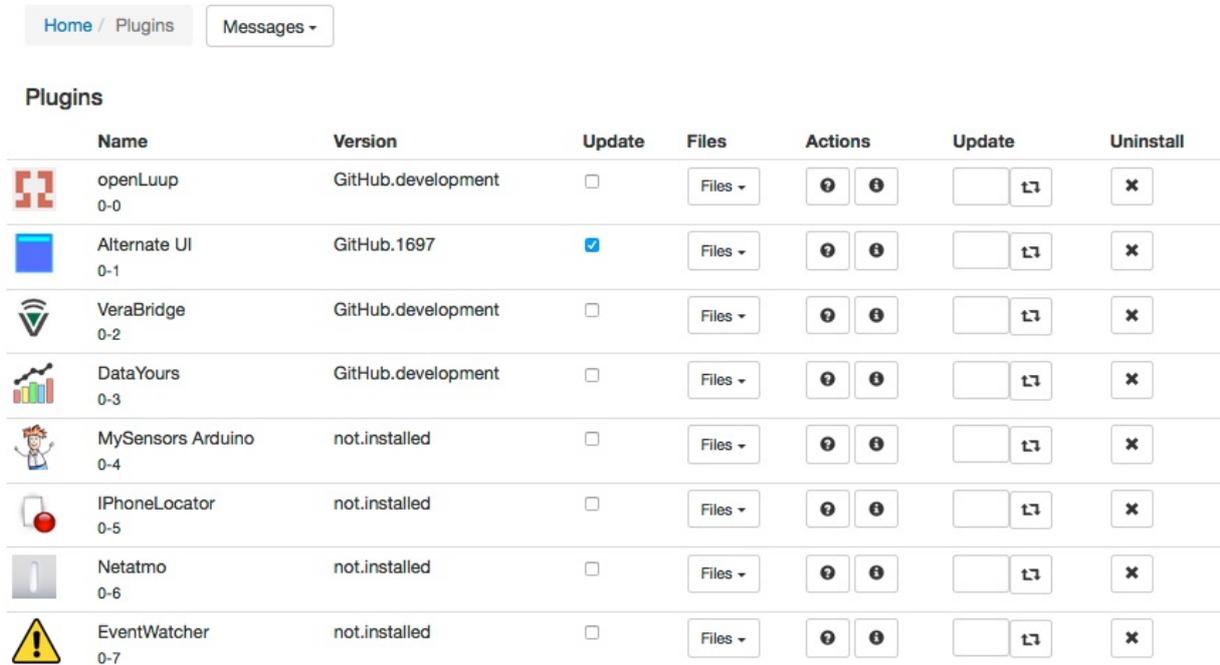
and browsing the reported URL will take you to the AltUI interface and show two devices: openLuup and AltUI. From here on, the interface can be used to configure the system.

3. Making openLuup run at system boot time

This initial running instance of openLuup will not persist between system reboots. To do that, some system-specific file editing is required. There are at least three options for doing this as described in the Appendix: Starting openLuup at system boot time.

4. Linking to Vera

Using the AltUI menus to navigate More > Plugins takes you to the plugin page:

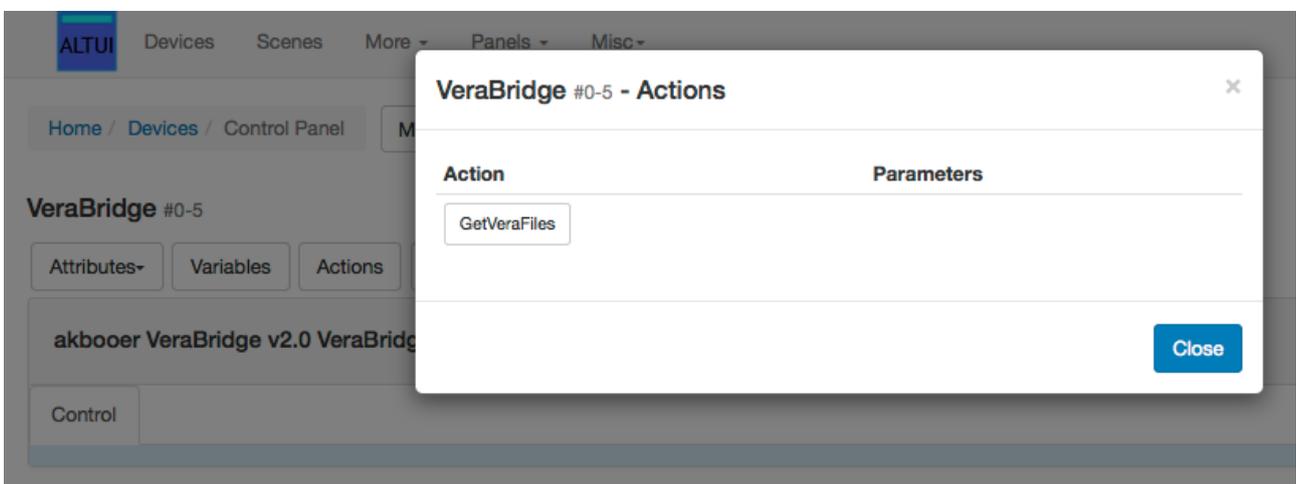


| Name | Version | Update | Files | Actions | Update | Uninstall |
|------------------------------------------------------------------------------------------------------------|--------------------|-------------------------------------|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
|  openLuup 0-0 | GitHub.development | <input type="checkbox"/> | Files - |   | <input type="text"/>  |  |
|  Alternate UI 0-1 | GitHub.1697 | <input checked="" type="checkbox"/> | Files - |   | <input type="text"/>  |  |
|  VeraBridge 0-2 | GitHub.development | <input type="checkbox"/> | Files - |   | <input type="text"/>  |  |
|  DataYours 0-3 | GitHub.development | <input type="checkbox"/> | Files - |   | <input type="text"/>  |  |
|  MySensors Arduino 0-4 | not.installed | <input type="checkbox"/> | Files - |   | <input type="text"/>  |  |
|  IPhoneLocator 0-5 | not.installed | <input type="checkbox"/> | Files - |   | <input type="text"/>  |  |
|  Netatmo 0-6 | not.installed | <input type="checkbox"/> | Files - |   | <input type="text"/>  |  |
|  EventWatcher 0-7 | not.installed | <input type="checkbox"/> | Files - |   | <input type="text"/>  |  |

Simply clicking on the Update button (located between Actions and Uninstall, **NOT the Update checkbox which you may safely ignore**) for VeraBridge will install that plugin.

You have to go to the Devices page and the Attributes tab of the new VeraBridge in order to enter the IP address of your Vera. Moving to the Misc > Reload Luup Engine menu will reload and the bridge should now have connected to that Vera and display its devices and scenes as 'clones' on openLuup.

There is just a little further configuration to do for bridged devices.



None of the device files or icons have yet been downloaded to openLuup, so the icons will be missing and also devices will not have their full functionality. The bridge allows an easy way to retrieve these files: simply go to the **Action** tag of the bridge's control panel and press the **GetVeraFiles** button.

This will copy all device files (D_xxx.xml, D_xxx.json, I_xxx.xml, S_xxx.xml, J_xxx.js, etc...) into **/etc/cmh-ludl/files/** and the icons (xxx.png) in **/etc/cmh-ludl/icons/** and reload the system. After a possibly required browser refresh, the device icons and full functionality should be in place.

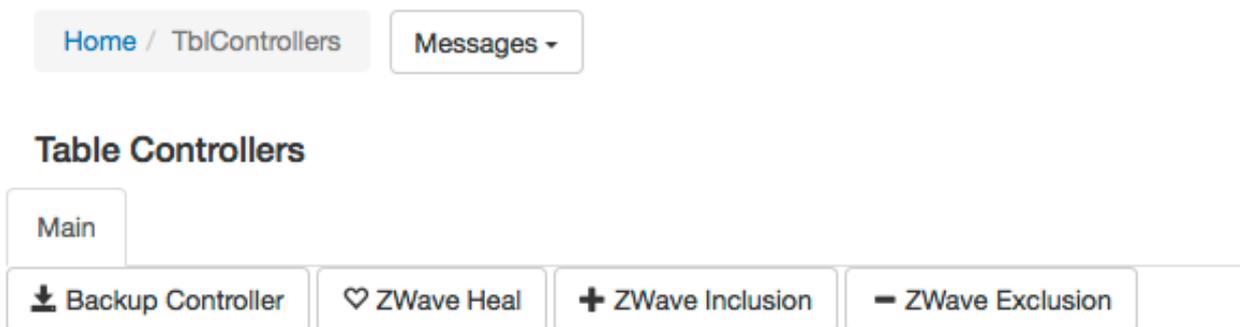
Read more about VeraBridge capabilities in a later section of this guide.

Backing up the system

AltUI makes it very easy to create a backup of your system configuration – all of which is defined in one (large) JSON-encoded file: **/etc/cmh-ludl/user_data.json**.

The More > Controllers page provides a button to save a copy to **/etc/cmh-ludl/backup/** in a file named something like **backup.openLuup-88800000-2016-05-01.lzap**. Repeated backups on the same day overwrite one another. Different days have unique names and are not automatically purged.

This file is a binary compressed version of the user_data.json file, and typically 5 to 6 times smaller. It may be used as a parameter to the openLuup_reload script and the initialisation process will uncompress it automatically.



Backup may also be initiated from Lua code (perhaps in a scheduled scene) with the single command

```
luup.inet.wget "/cgi-bin/cmh/backup.sh?"
```

More about VeraBridge

The VeraBridge plugin is an openLuup plugin which links to remote Veras. Unlike the built-in Vera 'bridge' capability (which, I think, uses UPnP discovery) this has no limitation as to the Vera firmware version that it links to, and you can quite happily run multiple copies of this linking to UI5 and UI7 remote machines (which, themselves, require no special software installation.)

VeraBridge provides local clones of devices and scenes from a remote Vera:

- reflects the status of the remote devices (ie. device variables and house mode)
- provides, through AltUI, a display panel and controls
- allows control of the remote devices (eg. on/off, dimming, ...) through the usual `luup.call_action` mechanism (or the control panel)
- makes remote scenes visible locally, allowing timers and triggers to be added
- maintains a local variable to reflect the remote machine's House Mode
- maintains a local variable showing the time of last contact with remote machine
- capability to lock the openLuup House Mode to the bridged machine or *vice-versa*.
- 'reverse bridging' or 'mirroring' of specified device variables: openLuup variables may be written to devices on the bridged Vera so that plugins and scenes there can be influenced by plugins and scene logic running under openLuup.
- the ability to download all device files and icons from the bridged Vera.

The device numbering is different from that on the remote machine, being incremented by multiples of 10000 for each different bridge, and one of VeraBridge's main functions is to intercept actions on the local devices and pass them to the remote ones. What does NOT work is to set a variable on a local device and expect that variable on the corresponding remote device to change.

Since it is possible (but fairly unlikely) that the VeraBridge will miss an update to a Vera device variable, every so often (actually, about every minute) it scans the whole lot and updates all the cloned devices on openLuup. This means that **WHATEVER** changes you make to cloned device variables will get overwritten frequently. The motto is this...
do NOT write variables to cloned devices.

Running under openLuup, VeraBridge supports *any* `serviceId/action` command request (ie. those *not* returning device status parameters) This covers most device control scenarios.

The bridge device also has a variable which reflects the remote House Mode (if the remote system is UI7 or greater) and which may therefore be used as a device variable watch trigger for scenes. Additionally, the bridge can 'mirror' the house mode in either direction, synchronising the mode of both (or more) machines.

Action calls with `serviceId urn:micasaverde-com:serviceId:HomeAutomationGateway1` made to the VeraBridge plugin are relayed to the remote Vera. This means that requests made to VeraBridge for actions like "SetHouseMode" and "SetGeoFence" will be effective on the bridged Vera.

1. Bridging to multiple Veras

Multiple bridges may be installed (through the Create button on the devices page.)

The first Vera linked to is special, for a number of reasons:

TBD...

2. Mirroring individual device variables from openLuup to Vera

Any openLuup device variable may be 'mirrored' to any bridged Vera. That is, a variable on a remote Vera device can actively track the value of a local variable.

Each instance of the VeraBridge plugin registers itself with AltUI as a Data Storage Provider with a name of the form **Vera@xxx.xx.xx.xx** where the 'x's represent the IP address of the remote Vera.

Simply going to AltUI's Device > Variables pane you can select to watch any variable by clicking on the 'graph' button, selecting the appropriate Vera to push the data to and enter the device number (on the remote machine) where you want that variable to be mirrored.

A device number on its own means that the current variable's serviceld and name will be used on the remote machine. Alternatively the following syntax may be used to force a specific serviceld / name:

device.serviceld.variable (eg. 42.urn:something:serviceld:newservice.NewName)

device.*.variable (eg. 42.*.NewName to retain existing serviceld)

3. Synchronising House Mode

The bridge has a device variable HouseModeMirror, which you can set to one of three values:

- "0" : no mirroring,
- "1" : local mirrors remote machine,
- "2" : remote mirrors local machine.

You have to reload to make this change effective. Because of Vera's built-in delay in changing HouseMode, mode 2 can take a while (typically 30 seconds or more) to kick in, but they should synchronise in the end.

Note that it's not easily possible to have this synchronisation work in both directions simultaneously because of a race condition that arises from the latency of Vera changing modes. You can, however, have openLuup mirror one bridged Vera's mode and have other bridged Veras which mirror that.

Also note that the openLuup plugin itself has a HouseMode variable which may be used as a variable watch trigger to kick off scenes when the house mode is changed (no need for a separate HouseMode plugin.)

Other configuration variables

My thanks to openLuup enthusiast @explorer for having made the original suggestion to add more customisation to VeraBridge: [Selecting devices in VeraBridge](#).

To quote from that post:

"I thought it would be nice to add some basic device filtering mechanism to VeraBridge, so I added these variables:"

- **ZWaveOnly** – if set to true then only Z-Wave devices are considered by VeraBridge.
- **IncludeDevices** – a comma-separated list of devices to include even if ZWaveOnly is set to true.
- **ExcludeDevices** – a comma-separated list of devices to exclude from synchronization by VeraBridge, takes precedence over the first two.

Also:

- **BridgeScenes** – enable/disable linking to remote scenes by setting to 'true'. If you have added or deleted some scenes on Vera and you wish to update the openLuup links, then you need to turn BridgeScenes off, reload, turn them on, and reload again.
- **CloneRooms** – set to true to force all bridged devices to be in the same rooms as on the remote Vera. Devices which are in "No Room" remotely, are placed in the appropriately named "MiOS..." room which corresponds to the remote machine name.

The Data Historian

openLuup includes a 'Data Historian' which stores previous device variable values in both in-memory cache and (optionally) on-disk archives. Taking lessons learned from DataYours and EventWatcher, also from AltUI's Data Storage Providers, it offers an almost configuration-free approach to storing (and retrieving) variable values. In addition, and in contrast to those other options and the dataMine plugin, it provides a natural way of reading historic variable values through the usual Luup interface. Also included is a Grafana Data Source API so that data may be easily visualised if you already have a Grafana installation, or plotted directly (using Google Charts.) A big plus here is that you don't have to have set up storage for any specific variable, they are (almost) all available.

In summary:

- only numeric variable values are stored
- by default, the in-memory cache stores all variable changes since system startup, with a small number of exceptions, retaining the most recent 1000 values (by default)
- variables not cached (by default) include timestamp-like ones, including LastUpdate, Polls, low-level Zwave, and some dates
- the on-disk archive is enabled by the single LuaStartup line `luup.attr_set ("openLuup.Historian.Directory","history/")`, where 'history/' is the path (including trailing '/') relative to `cmh-ludl/` where you want the data to reside.
- by default, only a small subset of cached variables are archived on disk, including security sensors Tripped, temperature, humidity, and generic sensors Current values, energy metering KWH and Watts, battery levels, openLuup system memory and cpu stats, ...
- the on-disk archives are implemented as Graphite Whisper files (as in DataYours)
- different data types have different sample rates and retention policies. For example, temperature data is sampled every 20 minutes, initially, but reduced in a number of stages to once per day after a year, and battery levels are just sampled once per day.
- default total on-disk archive duration is 10 years. Typical file sizes, one per variable (containing multiple resolution data) are about 300Kbyte.
- system impact is minimal, with high performance. The additional system load is less than writing to the logs.
- database maintenance effort is zero. The disk archive is of fixed size per variable, and total file space is roughly proportional to the number of archived variables.
- the openLuup Console has two pages to view historian activities: `openLuup > Historian` (for the in-memory cache), and `Files > History DB` (for the on-disk archives)
- the URL for the Grafana Data Source interface is simply your `openLuup:3480` port
- programmatically, data is retrieved using the `luup.variable_get()` call

On my 'production' system which is linked to 3 Veras, Netatmo, Philips Hue, and some MySensors Arduinos, there are over 4000 device variables. About 300 of those are cached in memory, and their on-disk archives take about 90 Mbyte. There are, on average, 12 updates per minute, each one taking about 1.5 mS on an RPi. (I don't have any security sensors triggering frequently on this system.)

For plugin developers, reading data from the cache/archive is trivial, using an additional {start,end} time parameter to `variable_get()`:

```
local v0,t0 = luup.variable_get ("urn:upnp-  
org:serviceId:TemperatureSensor1", "CurrentTemperature", 33)  
print ("normal current value", v0,t0)
```

```

local v2,t2 = luup.variable_get ("urn:upnp-
org:serviceId:TemperatureSensor1","CurrentTemperature", 33, {os.time()-3600,
os.time()})
print ("over the last hour", pretty {value = v2, times=t2})

```

gives

```

normal current value 31.3      1530026929
over the last hour  {
    times =
{1530023886,1530023926.5466,1530024526.9677,1530025127.1189,1530026327.7046,1
530026929.0275,1530027486},
    value = {31.1,31,31.2,31.3,31.2,31.3,31.3}
}

```

If the data over the requested time range is all in cache, then the time resolution will be better than 1mS. Once on disk, it depends on the retention policy of the particular archive (sensible defaults, but all configurable if necessary) never finer than 1 second, but usually 5-10 minutes. If there is no data within the time range, then arrays are empty, but within a non-empty array there are no nil points, the times simply denoting (approximately) when the variable changed.

DataYours has its own Whisper database, location defined by the contents of DataYours variable LOCAL_DATA_DIR, usually 'whisper/'. Data Historian has its data wherever you've set the system attribute openLuup.Historian.Directory, usually 'history/'. **You should not map these two directories to the same folder.**

Pointing Grafana at openLuupIP:3480/, will allow its metric menus to find both databases, but the metric trees will be merged. However, adding the following line to Lua Startup

```

luup.attr_set ("openLuup.Historian.DataYours", "whisper/")      --
overriding DY finder

```

Assuming that your LOCAL_DATA_DIR points to 'whisper/', then after a restart, you should see all your original DataYours file under a metric tree called DataYours...

```

"DataYours.Vera-88800000.008.urn^micasaverde-
com^serviceId^EnergyMetering1.EnergyUsage",
"DataYours.Vera-88800000.294.urn^micasaverde-
com^serviceId^EnergyMetering1.EnergyUsage",
"DataYours.Vera-88800000.386.urn^micasaverde-
com^serviceId^EnergyMetering1.EnergyUsage",
"DataYours.cpu.d",
"DataYours.memory.d",
"DataYours.unknown",
"DataYours.uptime.m",

```

...along with the other sub-trees generated by the historian (one for openLuup, and one for each bridged Vera.)

The user_data.json file

Configuration changes to the system happen in different ways in the three major phases of normal running, startup and shutdown:

STARTUP

- the default behaviour is to look for a JSON file called `user_data.json` in the current directory (`/etc/cmh-lud1/`) loading the device/room/scene configuration from that.
- an optional startup parameter is one of:
 - the word `reset`, which forces a 'factory reset', or
 - a `user_data` JSON formatted filename, or
 - a `.lzap` compressed backup file, or
 - a Lua filename to be run in the context of a factory-reset system (with no rooms or scenes, and only devices 1 (Gateway) and 2 (Z-wave controller) defined)

RUNNING

- every so often, the system configuration is check-pointed to the file `user_data.json`. This will capture device variable and attribute changes, scene creation/deletions, etc.
- logged events are written to the log files as they happen and not cached.

SHUTDOWN

- on `luup.reload`, or full exit, the configuration will be written to the file `user_data.json` in the current directory
- on `luup.reload`, or any other configuration change requiring a reload (eg. new child devices created) the process will exit with status 42
- on exit (from the HTTP request `id=exit`) will exit with status of 0 (successful exit)

This means that it is easy to save and restore arbitrary configurations, or start from scratch. Since any reload will cause the process to exit, it's necessary to launch `openLuup` from within a script, in order to restart automatically. A Unix shell script to do this, `openLuup_reload`, is included in the distribution (in the `openLuup - Utilities` directory) and shown here. For Windows, see the relevant Appendix.

```
#!/bin/sh
#
# reload loop for openLuup
# @akbooer, Aug 2015
# you may need to change 'lua' to 'lua5.1' depending on your install

lua openLuup/init.lua $1

while [ $? -eq 42 ]
do
    lua openLuup/init.lua
done
```

Whatever else you do, backup your functional user_data.json file.

Even after a system reboot, the only thing required to get `openLuup` restored to its previous state is to start the `openLuup_reload` script (with no parameters) since this just picks up the latest `user_data.json` file and runs with it.

More about the port 3480 HTTP server

As well as supporting Luup HTML requests, the openLuup HTTP server on port 3480 has some of the functionality of any normal web server, but also some special features:

1. File Access

The root directory for the port 3480 server is `cmh-ludl/` (or wherever the openLuup process is started.) However, in addition, the server will search in `../cmh-lu/` if it fails to find the requested file. This mimics Vera's use of that directory. It means that you can put `.xml`, `.json`, and other files there for convenience.

Note, however, that a plugin which expects to find a file in `/etc/cmh-ludl/` using `io.open()` will NOT automatically find it if it resides in `../cmh-lu/`.

The Lua package path has also been modified to search `../cmh-lu/` so that required Lua modules may also be located there. Note also that any files here are not in compressed form (`.lzo`) as they are in Vera.

Additionally, file-based icon references in `.json` files are redirected during loading to access the `cmh-ludl/icons/` directory through the port 3480 server. This means that the requests do not go from AltUI to port 80, but to port 3480, which in turn means that device icons work over a secure connection to port 3480 only. It also means that the icon directories under `/www/cmh/skins/default/...` are not required. Existing HTTP icon references are untouched.

2. index.html Files

The server looks for an `index.html` file if the request is for a directory (ends with `'/'`). This means that an `index.html` file may be used to present any web page you like, but in particular it may be used to redirect the request. For example:

With the following text in `cmh-ludl/index.html`

```
<!DOCTYPE html>
<html>
  <head>
    <!-- HTML meta refresh URL redirection -->
    <meta http-equiv="refresh"
      content="0; url=/data_request?id=lr_AltUI_Handler&command=home#">
  </head>
</html>
```

then `http://openLuupIP:3480/` redirects to AltUI.

Once again, this is useful functionality for access over a secure connection.

3. CGI processing

The openLuup HTTP server has a Lua Web Server API (WSAPI) connector to allow it to run Lua applications as CGIs. This is an industry-standard used on web servers such as Xavante, Apache, Lighttpd, and others.

This means that arbitrary web server functionality can be added to openLuup simply by writing a short, stand-alone, Lua module. The first time a CGI URL is accessed, the module is compiled and loaded. Thereafter, it runs just as quickly as any built-in server code. The API implementation adheres exactly to the (brief) documentation here: <http://keplerproject.github.io/wsapi/manual.html>.

The connector supports CGI code in the `cgi/` and `cgi-bin/` directories in `cmh-luudl/`. So, for example, taking *exactly* the code from the simple example in the above documentation:

```
#!/usr/bin/env wsapi.cgi

module(..., package.seeall)

function run(wsapi_env)
  local headers = { ["Content-type"] = "text/html" }

  local function hello_text()
    coroutine.yield("<html><body>")
    coroutine.yield("<p>Hello Wsapi!</p>")
    coroutine.yield("<p>PATH_INFO: " .. wsapi_env.PATH_INFO .. "</p>")
    coroutine.yield("<p>SCRIPT_NAME: " .. wsapi_env.SCRIPT_NAME .. "</p>")
    coroutine.yield("</body></html>")
  end

  return 200, headers, coroutine.wrap(hello_text)
end
```

...and putting it into `cmh-luudl/cgi/hello.lua`, allows you to invoke it with

```
http://openLuupIP:3480/cgi/hello.lua
```

...and get the returned page:

Hello Wsapi!

PATH_INFO: /

SCRIPT_NAME: /cgi/hello.lua

Any other CGI action can be implemented in the same way.

Note that shell scripts and other CGI language implementations are not (yet) supported by the port 3480 server, but this mechanism does enable some of the key CGIs used by Vera to be emulated through a small amount of effort.

Something about openLuup log files

On startup, openLuup writes its log data initially to `cmh-lud1/logs/Startup_LuaUPnP.log` until the initial devices and scenes are loaded and the Startup Lua has been run. This file is retained for the last 5 versions.

By default, openLuup writes its main log file to `cmh-lud1/log/LuaUPnP.log`. This location, and other logging parameters may be changed by defining openLuup attributes in the Startup Lua.

```
-- openLuup configuration options:
luup.attr_set ("openLuup.Logfile.Name", "logs/LuaUPnP.log") -- path to log file
luup.attr_set ("openLuup.Logfile.Lines", 2000)
luup.attr_set ("openLuup.Logfile.Versions", 5)
```

To keep file sizes manageable, it rotates the log from time to time (by default, after about 2,000 lines, which makes the file size about 250 Kb.) The five most recent log files are versioned with the suffix `.1 / .2 / ...etc.`, so you can go back and see what happened just before a reload. The reason for the reload should be one of the last lines in the log. Unlike Vera, this doesn't just happen randomly, but in response to either a user request or a device doing something which requires a reload (adding a new child device, for example.)

openLuup also maintains another log, which is a subset of the main one, and contains only device variable changes, scene invocations, and workflow messages. Those entries are written to a log file, if the `/var/log/cmh/` directory exists, in `/var/log/cmh/LuaUPnP.log`, with colour-highlighted variable names, as in Vera's log. The OS command that comes as standard in AltUI tails this log, and it's an easy way just to see the major changes happening in the system, without being cluttered by other activities which write to the log file. It is also used by AltUI to provide a history of device variable changes, scenes, and workflows. This log file is not versioned, and reset after 5000 lines, so it is possible to come to it sometimes and find it almost empty. This file is also essential for the variable and scene history functions of AltUI to operate – the file is scanned for the relevant variable changes which are presented as the change history under that button of the device variable. Since this file in openLuup is dedicated to this particular purpose, it's likely that variable and scene histories will be retained much longer than on a Vera.

openLuup doesn't use the same log level numbering system as Vera, trying to keep things simple. Log entries created by devices always carry their device ID. Here's a log entry made by device #7

```
luup_log:7: Arduino: urn:micasaverde-com:serviceId:HaDevice1,LastUpdateHR, 09:42, 42
```

If you're trying to troubleshoot a single plugin, then I always recommend that you do it in a system which only has that plugin. That way, almost all of the log entries are relevant. Of course, if it's complex multi-device logic, then you can't do that.

The log file name, path, length, and number of retained versions may be altered at startup. See **Appendix: Configuring key openLuup parameters at Startup**.

More about Scenes

Scene timers and actions can be defined with the AltUI interface in exactly the same way as on Vera. Triggers, however, are treated differently, because the Vera approach, itself based on UPnP definitions in various files, is deeply flawed and inflexible. Instead, the AltUI-supported mechanism of device variable triggers are used.

Using the appropriate button under the scene / trigger definition menu, you may select any device and variable as a trigger. The trigger will fire whenever the variable changes value, and using a small amount of Lua code, or the blocky graphical interface, you can construct arbitrary logic expressions combining, times, old and new variable values, other device variables, in fact almost anything you want, to determine whether or not the scene should actually execute.

openLuup also provides the capability to run a specified Lua function after the termination of an individual scene (scene finalisers), and, in addition, the ability to run shared, globally-defined functions at the start and end of all scenes (scene prolog and epilog code.)

Scenes on remote Veras may be linked to openLuup, or copied and saved to assist in the migration of automation logic.

1. VeraBridge and Remote Scenes

A Vera scene that has been linked to by VeraBridge is simply a perfectly normal openLuup scene (hence totally isolated from any scene code on your actual Vera) with one-line in the Lua code section (which you're able to view and edit) which fires off the remote scene.

This means you can add local triggers, timers, even other Lua code (before the the statement that runs the remote scene!) without any code impact on your actual Vera... very comforting when developing!

So, really, these types of scenes are not clones of the Vera scenes, but merely ways to trigger them remotely.

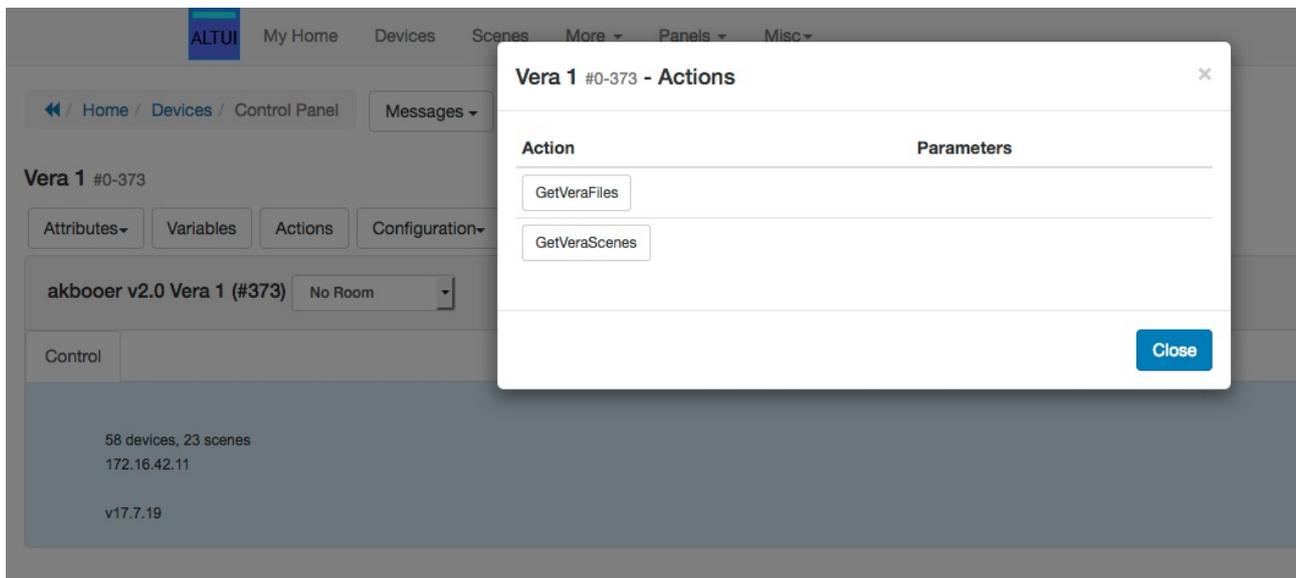
No aspect of the remote scenes can be changed – it is not an editor for remote scenes. You can't edit scenes from a remote Vera on openLuup if they are bridged by the VeraBridge plugin – you have to edit them on the original Vera.

They are really there as a basis for re-writing them to operate entirely locally in the openLuup environment as you migrate plugins and scene logic from Vera to openLuup. The bridge's capability to mirror openLuup variables to actual device variables on Vera makes this transition easier in some cases.

VeraBridge has a BridgeScenes variable. Set to **true** or **1** to enable linking to remote scenes, or anything else (including **false** or **0**) to disable this on startup.

2. Porting Vera scenes to openLuup

VeraBridge also has an action GetVeraScenes which creates **temporary** local copies of Vera scenes as an aid in migrating automation to openLuup.



Things to note:

- new scene numbering has 100,000 added to the local scene numbering
- new scenes name has TEMP COPY added as a suffix
- scenes are placed into the appropriate openLuup Vera room
- these scenes don't survive a reload (unless you explicitly renumber them)
- because of the above you can play to your heart's content and not damage anything
- the scenes are initialled PAUSED, so won't run
- all event triggers are redirected to local devices, but disabled
- any scenes with triggers have an extra one added, warning that they're unused!
- all device actions are redirected to local devices
- scene Lua is unchanged

All you have to do is go to the Actions tab for the VeraBridge you're interested in and press GetVeraScenes which runs a job (very quickly.) Switch immediately to the Scenes page and you should find them there.

To make a scene permanent, use the 'Copy' button on the Scene and it will create a new scene, with a low scene number, called 'Clone of XXX'. Don't forget to delete the original scene on Vera when you're ready to use the new openLuup one.

3. Scene Finalisers – Running Lua Code after Individual Scene Execution

Normal Lua scene code runs BEFORE any action in the scene. Scene finalisers allow you to run Lua code some time AFTER all the actions of a scene have executed.

In each scene that you want to do something, you return a SECOND parameter which is a function to be run when the scene 'finishes'. The default is 30 seconds after the START of the LAST delayed action (which may take an indeterminate time to run, depending on what it does.) There is an additional optional THIRD return parameter which overrides this default delay time in seconds.

So your scene Lua looks something like this:

```
local function finish() -- you can call it what you want
  -- things to do at the end go here
end

-- usual Lua code goes here

return status, finish, optional_delay
```

Be careful that EVERY return from the scene code looks like this last line. As usual, if 'status' is false then the scene (and the finaliser) will not run, if it's anything else then it will run. The simplest way to ensure this is to make the final return the ONLY return from the scene. If you can't do that you'll need to write things like this:

```
if something_special then
  return true, final
else
  return false
end

-- other things here perhaps

return true, final
```

Obviously, if you return false anywhere, the second parameter is not needed anyway.

4. Scene Prolog and Epilog Lua code – Global Functions

Thanks to constant pestering by forum member @DesT ;-), openLuup now has the ability to run global code as a pre/post for all scenes without having to edit them individually.

There are two parts to enabling these calls which can both be done in Startup Lua:

1) set the openLuup parameters which reference the calls

In Lua Startup define the global names for one or both of the procedures you want to call. For example:

```
luup.attr_set ("openLuup.Scenes.Prolog", "SCENE_Prolog")
luup.attr_set ("openLuup.Scenes.Epilog", "SCENE_Epilog")
```

The globals are shared in the scenes, Lua Startup, and Lua Test code environment.

2) define the global functions

```
function SCENE_Prolog (id)
    luup.log ("SCENE_Prolog: " .. id)
    -- any actions to run before all scenes go here
    return true -- or false to cancel
end

function SCENE_Epilog (id)
    luup.log ("SCENE_Epilog: " .. id)
    -- any actions to run after all scenes go here
end
```

The Prolog function is called before ANY scene code is executed. It's called with the scene id as a parameter. If it returns false then the scene will be aborted (as with normal Lua scene code.) Note that after this function is called (if it doesn't return false,) the regular scene Lua will be executed and that too may abort further scene execution.

The Epilog function is only called after a successful scene execution. It also receives the scene id as a parameter.

One useful application of the Prolog handler is that you can consolidate ALL your scene code into one place, in Lua Startup, without ever having to define any scene Lua when setting up the scene. Simply branch to an appropriate piece of code within the Prolog function to handle that particular scene number... or look up the scene name and branch based on that instead.

The possibilities are endless.

Appendix: Starting openLuup at system boot time

For openLuup to start up at system boot time, there are various approaches suggested by experts on the forum. Details are taken directly from the posts there:

- [openLuup Start on Bootup using Systemctl on Raspberry Pi by @groundglass](#)
- [/etc/rc.local, as used in turnkey systems by @CudaNet](#)
- [openLuup: init.d script by @martywendon](#)

I'm currently using the Systemctl approach...

systemctl with /etc/systemd/system/openluup.service

Thanks to @groundglass:

Another way to auto start openLuup on reboots using systemctl on a Raspberry Pi 3

I've had various degrees of successes (mainly unpredictable problems) with other projects on Raspberry Pi's with restarting programs with rc.local and crontabs. Typically it has related to when the network becomes available during the boot process. systemd has been more predictable for me. Here is how I set it up for openluup on a raspberry pi 3.

This assumes you installed openLuup at: /etc/cmh-ludl

Create a bash script to run openLuup in the background:

```
$ sudo nano /etc/cmh-ludl/run_openLuup.sh
```

Code:

```
#!/bin/bash
echo "Starting openLuup server"
echo "To see tail of logfile: tail -f ./out.log"
cd /etc/cmh-ludl
sudo rm ./out.log
nohup ./openLuup_reload >> out.log 2>&1 &
```

```
$ sudo chmod +x run_openLuup.sh
```

Next create the systemd service for openLuup setting it up to wait for network before starting

```
$ sudo nano /etc/systemd/system/openluup.service
```

Code:

```
[Unit]
Description=openLuup and AltUI Server for Vera 3
Wants=network.target
After=network.target
```

```
[Service]
Type=forking
WorkingDirectory=/etc/cmh-ludl
ExecStart=/bin/bash /etc/cmh-ludl/run_openLuup.sh
ExecStop=curl http://localhost:3480/data_request?id=exit

[Install]
WantedBy=multi-user.target
```

*Next (one time only) create the service and start openLuup. Make sure openLuup server is stopped using **http://openLuupIP:3480/data_request?id=exit***

```
$ sudo systemctl enable openluup
$ sudo systemctl start openluup
```

That's it. openLuup will now autostart with reboots.

Other useful commands:

```
$ sudo systemctl status openluup -l
$ sudo systemctl stop openluup
$ sudo systemctl disable openluup
```

/etc/init.d/openLuup

Thanks to @martynwendon:

So I whipped up a quick init.d script for openLuup, it works well for me on debian based systems.

To use it (some knowledge of Linux is assumed with the below commands):

```
create a file "openLuup" in /etc/init.d using your editor of choice
copy / paste script from below & save the file
make the file executable
test it, make sure you're happy with it (/etc/init.d/openLuup start)
execute "update-rc.d openLuup defaults" to activate it on start / stop of Linux
```

It supports "start/stop/restart/reload/kill"

Couple of notes:

I don't think openLuup is Linux "signal" aware (at least it didn't exit / restart cleanly when the process was sent signals). So it's not possible to use traditional stop / start / reload methods (as I understand it, it's very difficult to make Lua stuff signal aware, so this likely won't change). I used wget to call the exit / reload urls on localhost instead.

I like to run things like openLuup under a "screen" session as it means if they bomb out you should still be able to access the screen session and see any stdout / stderr fallout that may not make it into any log files. To resume the screen session use "screen -R -d -r openLuup" (will likely be a blank screen unless openLuup is outputting anything to stdout / stderr at the time).

Tail your log file using "tail -F /etc/cmh-ludl/LuaUPnP.log" - capital "F" makes tail follow the log file continuously and also re-open the file if it get's rotated.

Code:

```
#!/bin/sh
# Starts and stops openLuup
# /etc/init.d/openLuup
### BEGIN INIT INFO
# Provides:      openLuup
# Required-Start: $syslog
# Required-Stop:  $syslog
# Default-Start:  2 3 4 5
# Default-Stop:   0 1 6
# Short-Description:  openLuup
### END INIT INFO

#Load up openLuup when called
case "$1" in
start)
    echo "Starting openLuup.."
    cd /etc/cmh-ludl
    sudo screen -dmS openLuup Utilities/openLuup_reload
    echo "ok"

;;
stop)
    echo "Stopping openLuup.."
    sudo wget -q -t 1 -T 5 http://127.0.0.1:3480/data_request?id=exit
    echo "ok"

;;
kill)
    echo "Killing openLuup.."
    sudo screen -S openLuup -X quit
    echo "ok"

;;
restart)
    echo "Restarting openLuup.."
    $0 stop
    $0 start

;;
reload)
    echo "Reloading openLuup.."
    sudo wget -q -t 1 -T 5 http://127.0.0.1:3480/data_request?
id=reload
    echo "ok"

;;
*)
    echo "Usage: $0 {start|stop|reload|restart|kill}"
    exit 1

esac
```

/etc/rc.local

Thanks to @CudaNet and his turnkey systems guide:

> *Adjust startup(server) script for openLuup persistence....*

```
/etc/cmh-ludl$ sudo nano /etc/rc.local
```

> *You will see the following:*

```
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each multiuser runlevel.
# Make sure that the script will "exit 0" on success or any other
# value on error.
#
# In order to enable or disable this script just change the
# execution bits.
#
# By default this script does nothing.
```

> *Add these lines...*

```
sleep 20
cd /etc/cmh-ludl
./openLuup_reload

exit 0
```

> *IMPORTANT : 'exit 0' must NOT be removed.*

> *If using nano, press cntrl-o, press [enter], press cntrl-x to exit editor.*

Appendix: Configuring key openLuup parameters at Startup

There is a special top-level system attribute called “openLuup” (created at startup and not saved in the user_data.json file) which contains a table with some key system parameters some of which are user-configurable in the Lua Startup code.

The default structure includes:

```
{
  Backup = {
    Compress = "LZAP",
    Directory = "backup/"
  },
  Logfile = {
    Incoming = "true",
    Lines = 2000,
    Name = "logs/LuaUPnP.log",
    Versions = 5
  },
  Scenes = {
    Epilog = "",
    Prolog = ""
  },
  Status = {
    CpuLoad = "0.3%",
    Memory = "2.9 Mbyte",
    StartTime = "2018-01-27T11:32:53",
    Uptime = "0 days"
  },
  UserData = {
    Checkpoint = 60,
    Name = "user_data.json"
  },
  Version = "v18.1.18"
}
```

The key parts which can be changed include:

- openLuup.Backup.Directory — location of the backup files (path must exist already)
- openLuup.Logfile.Name — full path to log file
- ditto Lines and Versions — number of lines before log rotation, number of versions saved
- openLuup.UserData.Checkpoint — number of minutes between each user_data.json save.

Actually, the user_data.json file is saved after the first 6 minutes of the running system, and thereafter, every specified interval.

Some system internals may be inspected using the new console viewer UI (<http://openLuupIP:3480/console>.)



Appendix: Directory Structure and Additional Files

The installations steps 1, 2, 3, give you a functioning system, linked to a remote Vera. But to allow as many plugins as possible to run on openLuup, you may need additional files and directories.

1. Vera & openLuup directory structures and ancillary files

Vera's directory structure is quite complex, but the basic structure is shown below:

- /etc
 - /cmh
 - ui
 - /skins
 - /default
 - /icons/", "icons/" (UI5 icons)
 - /img
 - /devices/device_states/", "icons/" (UI7 icons)
 - /icons/", "icons/"
 - /cmh-lu
 - /cmh-ludl
- /usr
 - /bin
 - GetNetworkState.sh
- /var/log/cmh/LuaUPnP.log

openLuup is **designed to run in a non-privileged account using only a directory structure below its own home directory** to avoid file and directory permission problems (but see also next section.) The basic structure created by the install process is:

- /cmh-ludl (might be in /home/pi, or /etc/cmh, or anywhere else)
 - /backup
 - /files
 - /icons
 - /images
 - /logs
 - /openLuup
 - /trash
 - /www

Most plugins are completely agnostic as to file structures, simply being defined at startup by their device files. However, some advanced plugin authors do tricky things like re-writing JSON files and explicitly using the Vera file structure. It's possible to accommodate some of these by careful use of file aliases and extra directories. In other cases such authors have explicitly coded their plugins to discriminate between Vera and openLuup, and do the right thing.

Whilst openLuup's goal is to provide a run-time environment as similar as possible to Vera (but more reliable,) it's not fully achievable given the constraints of running on a third-party machine.

The AltAppStore plugin (an integral part of openLuup) also requires full access to the /tmp directory for temporary download files. This isn't usually an issue but there have been times when, for whatever reason, there have been permissions problems with this directory.

2. /usr/bin/GetNetworkState.sh and /etc/cmh/ui

Some plugins (eg. Sonos) use a shell command `/etc/bin/GetNetworkState.sh` to determine the machine's IP address. I used to provide this (actually for AltUI, but it doesn't need it now.) So you need to create that file with the contents shown in the Appendix.

On a Windows machine you may need to use the simpler hard-coded address version of the script shown there. I'm not actually sure that this works.

At least one plugin (IOSPush) requires the file `/etc/cmh/ui` with the single line (the digit seven):

7

3. openLuup_reload.bat for Windows

For Windows, you will need an alternative shell file to run the basic openLuup reload loop. I'm indebted to @vosmont for the following information which I've copied directly from the post here <http://forum.micasaverde.com/index.php/topic,34480.msg259829.html#msg259829>

For the moment, I use openLuup on Windows.

Install <http://luadist.org/> (contains Lua 5.1 and all the needed libraries)

Copy this file "openLuup_reload.bat" in "openLuup\etc\cmh-lud1", and change "LUA_DEV" according to LuaDist folder.

```
@ECHO OFF
SETLOCAL
SET LUA_DEV=D:\devhome\app\LuaDist\bin
SET CURRENT_PATH=%~dp0
ECHO Start openLuup from "%CURRENT_PATH%"
ECHO.
CD %CURRENT_PATH%
"%LUA_DEV%\lua" openLuup\init.lua %1

:loop
IF NOT %ERRORLEVEL% == 42 GOTO exit
"%LUA_DEV%\lua" openLuup\init.lua
GOTO loop

:exit
```

4. code for /usr/bin/GetNetworkState.sh

Some plugins (eg. Sonos, DLNA, Squeezebox, ...) require an external shell script (part of a standard Vera installation) to define the host machine IP address.

There's two basic ways to implement this.

This file can either be VERY simple:

```
echo -n 172.16.42.88
```

...with the IP address appropriately set (manually)

...or a more sophisticated approach using a Lua script to return the result (automatic) which is described here:

<http://forums.coronalabs.com/topic/21105-found-undocumented-way-to-get-your-devices-ip-address-from-lua-socket/>

```
#!/usr/bin/env lua
-----
-- discover main IP address of machine and write to standard output
-- http://forums.coronalabs.com/topic/21105-found-undocumented-way-to-get-your-
devices-ip-address-from-lua-socket/
-----
local socket = require "socket"
function myIP ()
    local mySocket = socket.udp ()
    mySocket:setpeername ("42.42.42.42", "424242") -- arbitrary IP/PORT
    local ip = mySocket:getsockname ()
    mySocket:close()
    return ip or "127.0.0.1"
end
io.write (myIP())
-----
```

Appendix: openLuup SMTP and POP3 (eMail) servers

SMTP server

openLuup (v18.3.14 and subsequent) includes a built-in SMTP server to receive email messages, Although included specifically for cameras which use this to trigger an associated motion detector device, this is a minimal implementation of an RFC 5321 compliant server with the LOGIN authentication method and no Transport Layer Security (TLS / SSL.)

The server only handles messages within the LAN sent to a specific TCP port (2525 by default) and does not relay them further, except to pass them to internal handlers, each of which may be registered to receive emails from a specific email address.

Out of the box, openLuup will start the SMTP server on port 2525. This can be changed in Lua Startup code with the following line:

```
luup.attr_set ("openLuup.SMTP.Port", 1234)    -- use port 1234 instead
```

The server comes with a number of predefined mailbox addresses:

- **postmaster@openLuup.local** – required for an SMTP compliant server
- **openLuup@openLuup.local** – general destination for openLuup messages. This mailbox performs no processing on incoming messages
- **images@openLuup.local** – used for camera trigger emails, or other messages with image attachments. Images are written to the image/ folder in the openLuup home directory.
- **test@openLuup.local** – all message data lines, including headers and body, will be written to the log to facilitate debugging. Beware of trying this on long messages!

This has been tested with a Foscam camera, which attaches three images per trigger event. The filenames are specified by the camera in the multipart ContentType header of the email message, which includes a clause like name="Snap_20180323-115529-0.jpg". Other cameras may use a different approach, but I'm more than happy to try and accommodate any different formats.

This feature brings with it the possibility of generating quite a large number of files, in due course. As a result I've found it helpful to include some sort of file retention policy implementation, so take a look at the openLuup File Management actions and variables.

Messages addressed to non-registered mailboxes will be rejected. In your own Lua Startup code or plugins, you can create a mailbox for your own code to access using the standard luup call:

```
luup_register_handler ("global_function_name", "email address")
```

for example:

```
luup_register_handler ("MyEmailHandler", "me@mymail.local")
```

Only one client can register for a specific email address, further attempts to register an existing mailbox will be rejected. However, the same handler can be used for multiple different addresses.

On receipt of an appropriately addressed email, your handler will be called thus:

```
function MyEmailHandler (email, data)
```

```
-- email is the registered mailbox address
-- you can have one handler for several addresses
-- data is the received data with a specific structure (see below)
end
```

The message table is a simple list of raw email data lines including headers and mail message body as received by the SMTP server. This is so that messages can be handled with the minimum amount of processing by the server.

However, for MIME-encoded messages (and almost ALL email clients use this format, including the LuaSocket library SMTP client module) there is a hidden utility function to decode headers and message (possibly a multipart message.)

The MIME format, as described in a plethora of internet standards, is quite complex with many parts of headers and messages encoded in different systems, mostly to handle legacy compatibility issues. The supplied decoder is fairly capable, but perhaps not totally comprehensive. It should deal with all normal message formats, including multipart messages with attachments using base64 and/or quoted-printable encoding and headers with RFC 2047 encoded words.

To decode a MIME message, simply write:

```
local message = data: decode ()
```

The decoded message format is of the form:

```
message = { header={...}, body=... }
```

- for a simple message, a body is a decoded string (from base64 or quoted-printable)
- for a multipart message, a body is a list of messages (possibly themselves with nested multiparts)
- header is BOTH an ordered list of decoded headers AND an index by header name (wrapped to lower case)

To understand the context of a body, you need to examine the headers. The header index makes this easy. To find, for example, the ContentType header (and if it exists at all) you could write:

```
local content = message.header["content-type"] -- indexed headers are lowercase
if content then
  -- whatever
end
```

To process parts of a multipart message you could write:

```
if type (message.body) == "table" then
  for _, part in pairs (body) do
    -- each part will have its own header and body
  end
end
```

It really is as simple as that.

Testing the SMTP server

You can manually test the SMTP server easily from any machine on the network (here, I'm doing it on the same machine)...

WITH NO AUTHORISATION

```
% nc 127.0.0.1 2525
220 (openLuup.smtp v18.3.24) [0.0.0.0] Service ready
helo from here
250 OK
mail from:akb@here
250 OK
rcpt to:test@openLuup.local
250 OK
data
354 Start mail input; end with <CRLF>.<CRLF>
this is a test message.
I hope it works!
.
250 OK
quit
221 (openLuup.smtp v18.3.24) [0.0.0.0] Service closing transmission channel
%
```

The all lower-case lines are user input. You have to mail TO: a recognised email address - the console SMTP server page will list those. The mailbox test@openLuup.local will write the raw data lines to the log.

The FROM: address can be anything of the form xxx@yyy.

Entries in the log file should be something like this:

```
2018-03-25 13:32:28.822 openLuup.smtp:: SMTP new client connection from
127.0.0.1: tcp{client}: 0x1a15448
2018-03-25 13:33:21.897 openLuup.smtp:: TO: test@openLuup.local
2018-03-25 13:33:21.898 luup.smtp.data:: Received: from (from here)
[127.0.0.1]
2018-03-25 13:33:21.898 luup.smtp.data:: by (openLuup.smtp v18.3.25)
[0.0.0.0];
2018-03-25 13:33:21.898 luup.smtp.data:: Sun, 25 Mar 2018 13:33:02 +0000
2018-03-25 13:33:21.898 luup.smtp.data:: this is a test message.
2018-03-25 13:33:21.898 luup.smtp.data:: I hope it works!
2018-03-25 13:33:21.898 openLuup.smtp:: EMAIL delivered to handler for:
test@openLuup.local
2018-03-25 13:33:25.313 openLuup.smtp:: SMTP QUIT received tcp{client}:
0x1a15448
```

WITH AUTHORISATION

Note the different client greeting EHLO instead of HELO, to which the server responds with a list of SMTP extensions, showing, in this case, that LOGIN authorisation is supported.

```
% nc 127.0.0.1 2525
220 (openLuup.smtp v18.3.26) [0.0.0.0] Service ready
ehlo from me
250 AUTH LOGIN
auth login
334 VXNlciBOYW11AA==          (base64 encoded Username: challenge)
myname                        (this should actually be base64 encoded)
334 UGFzc3dvcmQA             (base64 encoded Password: challenge)
mypassword                    (-ditto-)
235 Authentication successful.
```

... proceeds as previously

Any username and password will be accepted in response to the code 334 challenges.

For testing purposes, it may be useful to enable debugging of the openLuup.smtp module. This is achieved by inserting the following code in openLuup's Lua Startup code:

```
do -- SMTP debug
    local smtp = require "openLuup.smtp"
    smtp.ABOUT.DEBUG = true
end
```

A reload will be required to make this effective.

In debug mode, any mailbox destination will be accepted. It may also be useful to send test messages TO: test@openLuup.local. That mailbox will write the raw received data lines from the exchange to the log.

POP3 server

openLuup includes a full implementation POP3 server, along with some permanent mailboxes:

- `mail@openLuup.local` – a normal mailbox from which full messages (with any attachments) may be retrieved
- `events@openLuup.local` – only the subject line of any mail sent here will be retained

Together with the SMTP server, the POP3 server allows a fully internal LAN system to send, store, and retrieve email messages. It can conveniently be accessed by almost any email client. I send trigger messages from my camera to both `images@openLuup.local` which just stores the attached images in the `images/` folder, and also `events@openLuup.local` which lets me review any trigger times on an old iPod Touch.

Of course, if you want this to work when you are away from your home LAN, then you'd need to forward these via VeraAlerts or some such additional plugin.

Your email client can be used to manage the deletion of files from the mailboxes, or, alternatively, a timed scene using the openLuup `SendToTrash` and `EmptyTrash` actions will do the job automatically.

Appendix: Using Cameras with openLuup

I_openLuupCamera1.xml implementation file

A camera device created with this implementation file will create an associated child Motion Sensor device which is triggered when the camera's own motion detection algorithm sends an email.

Configuration

The camera's device implementation file may be set on the openLuup device's Attributes page, followed by a Luup reload. The only other significant parameters are the usual: ip attribute, and the URL and DirectStreamingURL device variables.

Camera configuration is obviously device-specific. For my Foscam camera (thanks to @Spanners) the important parameters are:

- **Enable** – ticked
- **SMTP Server** – the IP address of openLuup on your LAN eg. 172.16.42.156
- **SMTP Port** – 2525, or whatever other port number you configured in openLuup startup
- **Transport Layer Security** – none
- **Need Authentication** – No
- **SMTP Username / Password** – not used
- **Sender Email** – must include the form xxx@yyy, for example Foscam@Study.local
- **First Receiver** – openLuup@openLuup.local or images@openLuup.local

My camera (FI9831P) also sends three snapshots as email attachments. These are ignored if sent to openLuup@openLuup.local, but written to openLuup's images/ folder if sent to images@openLuup.local.

The Motion Sensor device will remain triggered for 30 seconds (or longer if the camera is re-triggered within that time.) In keeping with the latest security sensor service file, in addition to the Tripped variable, there is also an ArmedTripped variable which is only set/reset when the device is armed. This makes AltUI device watch triggers easy to write when wanting only to respond when the device is actually armed.

The **ArchiveVideo** action may be used to save single camera snapshots to openLuup's images/ folder. At present, the Format and Duration parameters are unused (since this is currently only implemented for single frames.)

Appendix: openLuup – Databases and Data Visualisation

External databases are useful for long-term storage, and visualisation, of sensor data and events.

Several plugins have been written specifically for Vera/openLuup to provide this functionality:

- dataMine – the original! Entirely Vera-based with possible USB storage. A self-contained eco-system with custom database format and graphics.
- DataYours – a Lua implementation of the Graphite / Whisper distributed database. Industry-standard protocols allow data to be sent to multiple databases. Includes a crude Google-charts powered graphics interface.

Since Vera-based storage is limited, and mounting USBs often challenging, CIFS (Common Internet File System) may be installed to access network-connected storage over the network. Whilst openLuup systems usually have massively more connected storage than Vera, the CIFS approach may also be used to good effect.

AltUI itself provides a facility called Data Storage Providers (DSP) which enables the sending of device variable data to remote databases, typically over HTTP. Currently, the default supported databases are: ThingsSpeak, Emoncms, and IfThisThenThat (IFTTT). It also provides a capability for users to extend the feature to additional databases.

openLuup builds on these tools to allow a number of further options:

- InfluxDB DSP – UDP connection to an externally configured InfluxDB database.
- Graphite DSP – UDP connection to an externally configured Graphite database.
- Graphite CGI – HTTP API to retrieve data from multiple dataMine & DataYours databases. (installed as part of the DataYours plugin)

The Graphite DSP is fully compatible with both DataYours and a real Graphite system. This means that you can send data to a remote DataYours installation *without* having DataYours installed on your local machine.

Whilst a number of these options provide their own data visualisations, the best experience for browsing and plotting data seems to be offered by third-party tools. Probably one of the best is Grafana which is able to access many of the above database systems through its own connectors. The Graphite CGI also allows Grafana to connect with data from DataMine and DataYours.

InfluxDB

My choice is for simplicity of use, which means configuring the UDP port of InfluxDB. Once installed, the InfluxDB configuration file is at `/etc/influxdb/influxdb.conf`. In that file, I have:

```
[[udp]]
enabled = true
bind-address = ":8089"
database = "openLuup"
retention-policy = ""
```

It's then a matter of configuring the Influx database (here called "openLuup") the way you want it. Note that the UDP port access can have its own default retention policy.

Finally, you need to arrange AltUI to give you InfluxDB as an option. It's just a matter of defining a system parameter to point to the UDP port in Startup Lua...

```
luup.attr_set ("openLuup.Databases.Influx", "172.16.42.129:8089")
```

For each watched variable you need just to specify a measurement name, optionally followed by any tags you want to do with it...

| Variable | Value |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| CpuLoad    | 0.8 |

Enable Push to :

- thingspeak
- influx**
- emoncms
- datayours

Influx

0-Measurement[,tags]

DataYours / Graphite

TO BE DONE.

Appendix: Sending and receiving UDP datagrams

Most communications protocols (HTTP, SMTP, POP3, ...) along with standard device plugin I/O are handled over the TCP connections. However, UDP offers a very lightweight, transaction-free communication mechanism used by some systems. DataYours, for example (a Lua implementation of the Carbon/Graphite database system) uses this for logging individual data metrics.

openLuup includes a mechanism to send UDP datagrams and also a callback to listen for incoming UDP datagrams arriving at specified ports.

The `luup.register_handler()` function has been extended to be able to specify the UDP protocol and a port to listen on. The handler is called with a similar parameter list to regular callback functions, but with slightly different semantics:

- `port` – the incoming port as a string, eg. "2222"
- `data` – a table with `{datagram = received_datagram_string, ip = sender_ip}`
- `protocol` – "udp", in this case

So, for example, in Lua Startup, or within a plugin you could write:

```
function myUDPcallback (port, data, protocol)
    print "Incoming UDP!"
    print (port)
    print ((json.encode(data)))
    print (protocol)
end

luup.register_handler ("myUDPcallback", "udp:2222")
```

The callback handler is called for each incoming datagram.

For sending, you simply need to open a UDP port on the destination IP:

```
local ioutil = require "openLuup.io" -- NOT same as luup.io or Lua's io.
local udp, err = ioutil.udp.open "123.4.5.67:2468"
if not udp then
    luup.log (err) -- some error message
end
```

and then use it whenever you like to send a datagram:

```
udp: send "body of UDP datagram"
```

Again, it's as simple as that.

Appendix: openLuup plugin Actions and Variables

The openLuup system itself runs a plugin called openLuup (always device #2) which offers useful variables and actions to assist in home automation tasks.

All of these elements are within the "openLuup" serviceId. Note that openLuup does not implement UPnP so does not use the Vera / MiOS / UPnP syntax "urn:xxx.com:serviceId:abcd1" for its own native services. Of course, for standard devices and third-party plugins, their own full serviceId should be used.

House Mode related actions and variables

When it comes to using house modes in Vera, you quickly run into two problems:

1. Mode is a system attribute, not a variable, so you can't use it as a trigger
2. SetHouseMode action is only defined for device #0 which you can't see from scenes

The openLuup plugin has its own HouseMode variable which tracks the system Mode attribute and may be used as a scene trigger.

It also has its own SetHouseMode action which may be selected from the list of scene actions for the openLuup plugin.

File Management actions and variables

action = "SendToTrash"

This action moves selected files to openLuup's trash/ folder, and has four parameters:

- **Folder** - the path of a folder below the openLuup home directory. For the application described here, this should be images, *but it must be a relative path name only.*
- **MaxDays** - the maximum age in days of files which should be retained
- **MaxFiles** - the maximum number of files that should be retained
- **FileTypes** - a string listing the types of files to which this applies, for example jpg gif bmp. It may also be set to * in which case, any file is applicable. Note that this does not act on subfolders or their contents.

One or both of MaxDays and MaxFiles may be specified. If either of the conditions is true, then the file is moved to trash.

Invoke this from Lua, like this:

```
luup.call_action ("openLuup", "SendToTrash"  
  {Folder="backup", MaxDays="", MaxFiles="10", FileTypes="lzap"}, 2)
```

Certain folders are protected from file management actions, in particular:

openLuup, cgi, cgi-bin, cmh, historian, whisper, files, icons, www

Moving files to the trash folder is reversible (manually!) but to delete them permanently, you may use the following action.

action = "EmptyTrash"

This action has one parameter:

- **AreYouSure** - needs to have the value "yes" (case-insensitive) to delete all the files in openLuup's trash/ folder.

The purpose of the AreYouSure parameter is to avoid accidental clicks on the EmptyTrash button on the device actions page having any effect.

The above actions are easily included into scheduled scenes which might, for example, run the SendToTrash action early in the morning every day, and the EmptyTrash every weekend.

AltUI-related variables

AltUI has its own actions and services with the serviceld of "urn:upnp-org:serviceld:altui1".

openLuup uses several of these variables to display information on its device panel, as do some related plugins like AltApStore and VeraBridge.

DisplayLine1 and DisplayLine2 are the relevant device variables.

System Status Variables

Global system parameters:

- MemAvail_Mb
- MemFree_Mb
- MemTotal_Mb

openLuup system parameters:

- CpuLoad – the percentage of CPU time being used by all of openLuup
- Memory_Mb – the total amount of memory that all of openLuup is using
- Uptime_Days – just how long we've been running
- Version – the system version as tagged in GitHub
- StartTime – the time of the last Luup restart

Appendix: Undocumented features of Luup

The documentation for Luup is poor: out of date, misleading, incomplete, unclear, and sometimes just plain wrong. However, “hats off” to those stalwart members of the forum who have contributed to the Wiki. During the implementation of openLuup a number of undocumented ‘features’ have come to light. In some cases, these features are used by various plugins, either deliberately or unknowingly, through sins of commission or omission. As a result, I’ve had to include them in the openLuup implementation.

If you’re a developer, please try NOT to rely on these things in your plugin code.

Here’s what I’ve found - if you know more, let me know.

- **lu_device** – this variable is often included in callback function parameter lists to indicate the target device, and that’s fine. However, it ALSO turns out to be in scope in the whole body of a plugin’s Lua code. Some plugins rely (possibly inadvertently) on this feature.
- **nil device parameter in luup.variable_watch** – whilst the use of a nil variable parameter to watch ALL service variables is documented, the use of a nil DEVICE is not, but works as expected: the callback occurs a change in ANY device with an update to the given serviceid and variable (thanks for @vosmont for that information.)
- **nil device parameter in luup.variable_set/get** – it appears that when called from device code, a missing device parameter gets the current luup.device variable value substituted.
- **urn:micasaverde-com:serviceid:HaDevice1, HideDeleteButton** – according to @reneboer: *“If you set that to 1 on UI7 it will not show the delete button at the bottom of the device Control Panel. I now use this to hide that on the child devices that are under full control of the parent device, including proper deletion.”*
- **the ordering of the <files> tag and the <functions> tag matters** – in implementation files, they are concatenated in that order and it matters to the scope of local variables defined there (thanks to @logread and @cybrmage for that nugget.)
- **only devices with a device file appear in the status request response** – what this means is that if you create a ‘dummy’ device on Vera, just to hang variables off, or as a holder for VeraBridge mirrored variables, those device variable values won’t appear on a bridged openLuup system *unless* you create the dummy device with a valid device file (just a device type will not do.) I usually use `D_CombosDevice1.xml` for this.
- **luup.inet.wget()** – returns a third parameter, which is the *actual* http.request status code.
- **luup.io.intercept()** – returns a boolean with complex meaning. See post by @a-lurker here: <http://forum.micasaverde.com/index.php/topic,48814.0.html>
- **/data_request?id=static** – This is just a subset of the &id=user_data request, which contains the static data. It’s the opposite of &id=user_data&ns=1, which suppresses the static data.
- ...

Appendix: Unimplemented features of openLuup

openLuup is, as of this time, an unfinished work. The following features are known to be unimplemented, poorly implemented, or non-functional, for a variety of reasons.

1. unimplemented luup API calls

- **luup.devices_by_service** – the definition of the functionality here [Luup Lua Extensions](#) is not adequate to make an implementation.
- **luup.job_watch** – not yet implemented.
- **luup.require** – undocumented.
- **luup.xj** – undocumented.

2. unimplemented HTTP requests

- **id=scene** – only create, delete, list, and rename are implemented (ie. no interactive creation of scenes.) Also note that scene triggers (or notifications) are stored, but not used. Any scene with a defined trigger will have one additional trigger added with a warning that these triggers are un-implemented.
- **id=action** – the virtual category 999 is not implemented. Actions on groups of devices defined by category is not implemented.
- **id=finddevice** – not implemented.
- **id=resync** – not implemented.
- **id=archive_video** – only implemented for single snapshots.
- **id=jobstatus** – not implemented.
- **id=relay** – not implemented.

Appendix: Differences between openLuup and Vera / MiOS

Features of Vera / MiOS not in openLuup

Hard to give an exhaustive list, but notable omissions include:

- UPnP triggers as defined in service files are not implemented.
- openLuup does not support events as published in the json files.
- openLuup does NOT use the system HTTP port 80 server, but implements its own port 3480
- the default user_data.json checkpoint time every 60 minutes, not 6.
- ...

Features of openLuup not in Vera / MiOS

The original goal for the development of openLuup was to have a system which was to be as indistinguishable as possible from a real Vera, aside, of course, for performance and reliability.

However, there are some blatantly obvious omissions from Vera's Luup which are very much 'nice to have' and don't break forward compatibility when moving from Vera to openLuup.

lul_scene:

In device implementation code, there is a variable **lul_device** which defines the current device number to the code. This may be used to index the **luup.devices** table for more information. However, there is no equivalent when executing scene code.

The variable **lul_scene** is now available to fill that gap, indicating the active scene number which may, in turn, be used to index the **luup.scenes** table to extract other information such as the name (description) of the current scene.

luup.call_timer():

In Vera, this call is one-shot and if you want repeating timers you have to call it again within the callback. In openLuup, there is an extra boolean parameter which, if true, reschedules the callback automatically. Important, if using this, NOT to call it again in the callback handler!

scene finalisers, prolog and epilog code:

See the relevant section above on these topics.

ShutdownCode:

There is an additional global attribute in the user_data called ShutdownCode. This may be set at startup, or other times, with `luup.attr_set ("ShutdownCode", "-- Lua code goes here")` and defines Lua code which is run prior to system shutdown (or restart.) This has been used to tidy up connections to external devices and servers prior to disconnection.