Lua Flash Store (LFS)

Work in Progress

The next PR will add extra LFS functionality and require some major changes to this document. Hence we will be updating this document in the next few weeks.

Some Acronym Definitions

Before I begin, I think it useful to define some acronyms in the context used in this paper. There are others like IoT which you might know and on first use are hyperlinked to an external (typically Wikipedia) definition.

Acronym	What it means in this paper
LFS	*Lua Flash Store*, and if you want to understand more then read on.
RTS	The Lua environment on the ESP chipset include a core Runtime System includes a VM tha
ESP	Espressif Systems Processor. A range of SoC devices based on an Xtensa core. NodeMCU
RAM	Random Access Memory that can be directly read or written by the CPU and can be used t
ROM	Read Only Memory, though for the purposes of this paper, this also includes a region of Fla
GC	Garbage Collection refers to the automatic management resources to collect dead objects

Background

Lua was originally designed as a general purpose embedded extension language for use in applications run on a conventional computer such as a PC, where the processor is mounted on a motherboard together with multiple Gb of RAM and a lot of other chips providing CPU and I/O support to connect to other devices.

An ESP8266 module is on a very different scale: it costs a few dollars; it is postage stamp-sized and only mounts two main components, an ESP SoC and a flash memory chip. The SoC includes on-chip RAM and also provides hardware support to map part of the external flash me v: release
a ROM addressable region so that the firmware code can be executed via an L1 cache out of this memory.

ESP SoCs also adopt a type of modified Harvard architecture found on many IoT devices where separate address regions are used for RAM code and RAM data and ROM code. ESPs also allow data constants to be loaded from code memory. The ESP8266 has 96 Kb of data RAM, but half of this is used by the operating system, for stack and for device drivers such as for WiFi support; typically **44 Kb** RAM is available as heap space for embedded applications. By contrast, the mapped flash ROM region can be up to **960 Kb**, that is over twenty times larger. Even though flash ROM is read-only for normal execution, there is also a "back-door" file-like API for erasing flash pages and overwriting them (though some care has to be taken to ensure cache integrity after update).

Lua's design goals of speed, portability, small kernel size, extensibility and ease-of-use make it a good choice for embedded use on an IoT platform, but with one major limitation: the standard Lua RTS assumes that both Lua data *and* code are stored in RAM, and this is a material constraint on a device with perhaps 44Kb free RAM and 512Kb free program ROM.

The LFS feature modifies the Lua RTS to support a modified Harvard architecture by allowing the Lua code and its associated constant data to be executed directly out of flash ROM (just as the NoceMCU firmware is itself executed). This enables NodeMCU Lua developers to create Lua applications with a region of flash ROM allocated to Lua code and read-only constants. The entire RAM heap is then available for Lua read-write variables and data for applications where all Lua is executed from LFS.

The ESP architecture provides very restricted write operations to flash memory: any updates use NAND flash rules, so any updates typically first require bulk erasing of complete 4Kb memory pages. This makes it impractical to modify Lua code pages on the fly safely, and hence LFS uses a "reflash and restart" paradigm for reloading the LFS region.

If you are just interested in learning how to quickly get started with LFS, then please read the chapters referenced in the Getting Started overview. The remainder of this paper is for those who want to understand a little of how this magic happens, and gives more details on the technical issues that were addressed in order to implement this feature in the following sections:

- Configuring LFS
- An Overview of LFS Internals
- Programming Techniques
- Compiling and Loading LFS Images

Configuring LFS

The LFS region can be allocated during firmware build:

- For Lua developers that prefer the convenience of our Cloud Build Service, the menu dialogue offers a range of drop down options to select the size of LFS region required.
- Some advanced developers might want to use Docker or their own build environment as per our Building the firmware documentation, and in this case the file `app/include/user_config.h` is used and includes inline documentation on how to select the configuration options to make an LFS firmware build.

You also have the option to reconfigure the LFS and SPIFFS regions at any time (typically after restart). See the function `node.getpartitiontable()`, so for example executing the following on one of my test modules:

```
do
  local s,p={},node.getpartitiontable()
  for _,k in ipairs{'lfs_addr','lfs_size','spiffs_addr','spiffs_size'} do
    s[#s+1] ='%s = 0x%06x' % {k, p[k]}
  end
  print ('{ %s }' % table.concat(s,', '))
end
```

prints out:

```
{ lfs_addr = 0x096000, lfs_size = 0x010000, spiffs_addr = 0x100000, spiffs_size = 0x100000 }
```

which just the format of the array parameter for `node.setpartitiontable()`. By default the LFS region starts immediately after the firmware so bringing this forward can cause errors. Likewise I usually start my SPIFFS on the 1Mb boundary and only allocate what I think that I need as having a larger SPIFFS partition slow I/O performance. However if I want to increase the partition, say to 128Kb then I can just patch this into a set partition command:

```
> node.setpartitiontable{ lfs_addr = 0x096000, lfs_size = 0x020000,
>> spiffs_addr = 0x100000, spiffs_size = 0x100000 }
```

restarts the CPU and after startup, executing the above `do` `end` fragment returns:

```
{ lfs_addr = 0x096000, lfs_size = 0x020000, spiffs_addr = 0x100000, spiffs_size = 0x100000 }
```

Job done.

An Overview of LFS Internals

LFS Internal Structure

v: release

Whilst memory capacity isn't a material constraint on most conventional machines, the Lua RTS still includes some features to minimise overall memory usage. In particular:

- The more resource intensive data types are know as *collectable objects*, and the RTS includes a GC which regularly scans these collectable resources to determine which are no longer in use, so that their associated memory can be reclaimed and reused.
- The Lua RTS also treats strings and compiled function code as collectable objects, so that these can also be GCed when no longer referenced

The compiled Lua code which is executed by the RTS internally comprises one or more function prototypes (which use a `Proto` structure type) plus their associated vectors (constants, instructions and meta data for debug). Most of these compiled constant types are basic (e.g. numbers); strings are the only collectable constant data type. Other collectable types such as arrays are actually created at runtime by executing Lua compiled code to build each resource dynamically.

Overlay techniques can be used ensure that active functions are loaded into RAM on a just-in time basis and thus mitigate RAM limitations. Moving Lua "program" resources into flash ROM typically at least doubles the effective RAM available, and typically removes any need to complicate applications code by implementing overlaying.

When any Lua file is loaded (without LFS) into an ESP application, the RTS loads the corresponding compiled version into RAM. Each compiled function has an associated own Proto structure hierarchy, but this hierarchy is not exposed directly to the running application; instead the compiler generates `CLOSURE` instruction which is executed at runtime to bind the Proto to a Lua function value thus creating a closure. Since this occurs at runtime, any Proto can be bound to multiple closures. A Lua closure can also have multiple Upvalues bound to it, and so function value is an updatable object in that it is referring to something that can contain writeable state, even though the Proto hierarchy itself is intrinsically read-only.

As read-only resources that are store in flash ROM clearly cannot be collected by the GC, such ROM based resources cannot reference any volatile RAM-based data elements, though the converse can apply: updatable resources in RAM can reference read-only ones in ROM.

All strings are stored internally in Lua with a header structure known as a `TString`, In Lua 5.1, all TStrings were interned, so that only one copy of any string is kept in memory, and most string manipulation uses the address of this single copy as a unique reference. Lua 5.3 divides strings into **Short** and **Long** subtypes with short strings being handled in the same way as Lua 5.1. In contrast, long strings are created and copied by reference, but are *not* guaranteed to be stored uniquely. Guaranteeing uniqueness requires the string to be hashed and this can have a runtime overhead for long strings, yet identical long strings are rarely generated by other than by copy-reference; hence the general runtime savings for not hashing long strings exceed the small chance of storage duplication.

All of this complexity is hidden from running Lua applications, but does impact our LFS implementation. The Lua global state links to two (short) string tables: `ROstrt` in LFS for short strings stored in LFS; and `strt` for short strings in RAM. Maintaining integrity across these two string tables at runtime is simple and low-cost, with LFS resolution process extended across both the RAM and ROM string tables. Hence any strings already in the ROM string table can be reused and this avoids the need to add an additional entry in the RAM table. This significantly reduces the size of the RAM string table, and also removes a lot of strings from the LCG scanning.

Lua GC of both types is essentially the same (and skipped for LFS strings) except that collection of long strings does not need to update the `strt`.

LFS internal layouts are different for our Lua 5.1 and Lua 5.3 version execution environments, with the Lua 5.3 version:

- Reflecting the changes to the Lua core that support the new Lua 5.2 and 5.3 language features (such the two string subtypes);
- Facilitating the additions of a second LFS region to allow separate System and Application LFS regions; this will also add a second `RO2strt`;
- Facilitates the on-ESP building of LFS regions at runtime, thus optionally removing the need for host-based code compilation;
- Improving the scalability of LFS function resolution.

Any Lua file compiled into the LFS image includes its main function prototype and all the child resources that are linked in its `Proto` structure, so all of these resources are compiled into the LFS image with this entire hierarchy self-consistently within the flash memory:

- A TString reference to the name of the function
- The vector of Lua VM instructions used to execute the function codes
- (optional) A packed map of instruction offset -> line number used for error tracebacks
- A vector of constants used by the function
- (optional) metadata on local variables
- metadata on local variables (some fields optional)
- a vector of references to other Protos compiled within the current one.

The optional fields are include or not depending on the stripdebug setting at the time c v: release
compilation.

Lua 5.1 LFS images include a (hidden) Lua index function which has an `if n == "moduleN" then return moduleN end` chain to resolve functions within the LFS. This as been replaced by a ROTable in Lua 5.3 significantly reducing lookup times for larger LFS images.

Lua uses a binary tokenised format for dumping compiled Lua code into a file based "compiled" format. Unlike the loading of Lua source which involves on-demand compilation at runtime, "undumping" compiled Lua code is the reverse operation to "dump" and is 5-10× faster.

With Lua 5.3 LFS image file formats are a minor extension to the dump format and loading the image file into an LFS is a variant of the "undump" process (which shares the same internal functions).

Impact on the Lua Garbage Collector

The GC applies to what the Lua VM classifies as collectable objects (strings, tables, functions, userdata, threads -- known collectively as `GCObjects`). A simple two "colour" GC was used in previous Lua versions, but Lua 5.1 introduced the Dijkstra's 3-colour (*white*, *grey*, *black*) variant that enabled the GC to operate in an incremental mode. This permits smaller GC steps interspersed by pauses, and this is very useful for larger scale Lua implementations. Even though this incremental mode is less useful for small RAM IoT devices, NodeMCU retains this standard Lua implementation.

In fact, two *white* flavours are used to support incremental working (so this 3-colour algorithm really uses 4). All newly allocated collectable objects are marked as the current *white*, and a link in `GCObject` header enables scanning through all such Lua objects. Collectable objects can be referenced directly or indirectly via one of the Lua application's *roots*: .

The standard GC algorithm is quite complex and assumes that all GCObjects are in RAM and updatable. It operates in two broad phases: **mark** and **sweep** where the mark phase does a recursive walk starting at the GC *roots* (the global environment, the Lua registry and the stack) and marks the collectable objects in use. This process is complicated by the fact that the collector is incremental, that is its processing can be broken into packets of collection that can be interleaved within normal memory allocation actions. Once a mark phase has been completed, the sweep phase chains down the linked list of GCobjects to detect unmarked objects and reclaim them. Because LFS ROM GCObjects are treated as immutable, GC processing can still maintain overall object integrity, with the LFS modifications preventing the marking updates to ROM GCObjects (since any attempt to update any ROM-base structure during GC will result in the firmware crashing with a memory exception).

There are all sorts of nuances needed because of the incremental nature and to allow L v: release
finalizers to take an active role in collection, and these finalizers can themselves trigger allocation actions. Not for the faint hearted. However, ROM GCOjects can still be handled

robustly within this scheme because: - Whilst a RAM object can refer to a ROM object, the converse in _not_true: a ROM object can never refer to a RAM one. Hence the recursive mark phase can safe abort its recursive walk at any node when a ROM object is detected. - ROM objects are in a separate linked list to that used by the sweep process on RAM objects and so are never swept.

Lastly note that the cost of GC is directly related to the size of the GC sweep lists. Therefore moving resources into LFS ROM removes them from the GC scope, and therefore reduces GC runtime accordingly.

Programming Techniques

A number of developers have commented that moving applications into LFS simplifies coding style, as you need to be far less concerned about dynamic loading of code just-in-time to mitigate RAM use, and source modules can be larger if that suits application structure. This makes it easier to adopt a clearer coding style, so ESP Lua code can now looks more similar to host-based Lua development. Also Lua code can still be loaded from SPIFFS, so you still have the option to keep test code in SPIFFS, and only move modules into LFS once they are stable.

Accessing LFS functions and loading LFS modules

The main interface to LFS is encapsulated in the `node.LFS` table. See `lua_examples/lfs/_init.lua` for the code that I use in my `_init` module to do create a simple access API for LFS. There are two parts to this.

The first sets up a table in the global variable `LFS` with the `__index` and `__newindex` metamethods. The main purpose of the `__index()` is to resolve any names against the LFS using a `node.flashindex()` call, so that `LFS.someFunc(params)` does exactly what you would expect it to do: this will call `someFunc` with the specified parameters, if it exists in in the LFS. The LFS properties `_time`, `_config` and `_list` can be used to access the other LFS metadata that you need. See the code to understand what they do, but `LFS._list` is the array of all module names in the LFS. The `__newindex` method makes `LFS` readonly.

The second part uses standard Lua functionality to add the LFS to the require package.loaders list. (Read the link if you want more detail). There are four standard loaders, which the require loader searches in turn. NodeMCU only uses the second of these (the Lua loader from the file system), and since loaders 1,3 and 4 aren't used, we can simply replace the 1st or the 3rd by code to use `node.flashindex()` to return the LFS module. The supplied `_init` puts the LFS loader at entry 3, so if the module is in both SPIFFS and LFS, then the SPIFFS version will be loaded. One result of this has burnt me during development: if there is an out of date version ii v: release
then it will still get loaded instead of the one if LFS.

If you want to swap this search order so that the LFS is searched first, then SET `package.loaders[1] = loader_flash` in your `_init` code. If you need to swap the search order temporarily for development or debugging, then do this after you've run the `_init` code:

```
do local pl = package.loaders; pl[1],pl[3] = pl[3],pl[1]; end
```

Moving common string constants into LFS

LFS is mainly used to store compiled modules, but it also includes its own string table and any strings loaded into this can be used in your Lua application without taking any space in RAM. Hence, you might also want to preload any other frequently used strings into LFS as this will both save RAM use and reduced the Lua Garbage Collector (**GC**) overheads.

The new debug function `debug.getstrings()` can help you determine what strings are worth adding to LFS. It takes an optional string argument `'RAM'` (the default) or `'ROM'`, and returns a list of the strings in the corresponding table. So the following example can be used to get a listing of the strings in RAM.

```
do
  local a=debug.getstrings'RAM'
  for i =1, #a do a[i] = ('%q'):format(a[i]) end
  print ('local preload='..table.concat(a,','))
end
```

You can do this at the interactive prompt or call it as a debug function during a running application in order to generate this string list, (but note that calling this still creates the overhead of an array in RAM, so you do need to have enough "head room" to do the call).

You can then create a file, say `LFS_dummy_strings.lua`, and insert these `local preload` lines into it. By including this file in your `luac.cross` compile, then the cross compiler will also include all strings referenced in this dummy module in the generated ROM string table. Note that you don''t need to call this module; it's inclusion in the LFS build is enough to add the strings to the ROM table. Once in the ROM table, then you can use them subsequently in your application without incurring any RAM or GC overhead.

A useful starting point may be found in lua_examples/lfs/dummy_strings.lua; this saves about 4Kb of RAM by moving a lot of common compiler and Lua VM strings into ROM.

Another good use of this technique is when you have resources such as CSS, HTML and JS fragments that you want to output over the internet. Instead of having lots of small resource files, you can just use string assignments in an LFS module and this will keep these con v: release LFS instead.

Choosing your development life-cycle

The build environment for generating the firmware images is Linux-based, but you can still develop NodeMCU applications on pretty much any platform including Windows and MacOS, as you can use our cloud build service to generate these images. Unfortunately LFS images must be built off-ESP on a host platform, so you must be able to run the `luac.cross` cross compiler on your development machine to build LFS images.

- For Windows 10 developers, one method of achieving this is to install the Windows Subsystem for Linux. The default installation uses the GNU `bash` shell and includes the core GNU utilities. WSL extends the NT kernel to support the direct execution of Linux ELF images, and it can directly run the `luac.cross` and `spiffsimg` that are build as part of the firmware. You will also need the `esptool.py` tool but `python.org` already provides Python releases for Windows. Of course all Windows developers can use the Cygwin environment as this runs on all Windows versions and it also takes up less than ½Gb HDD (WSL takes up around 5Gb).
- Linux users can just use these tools natively. Windows users can also to do this in a linux VM or use our standard Docker image. Another alternative is to get yourself a Raspberry Pi or equivalent SBC and use a package like DietPi which makes it easy to install the OS, a Webserver and Samba and make the RPi look like a NAS to your PC. It is also straightforward to write a script to automatically recompile a Samba folder after updates and to make the LFS image available on the webservice so that your ESP modules can update themselves OTA using the new `HTTP_OTA.lua` example.
- In principle, only the environment component needed to support application development is `luac.cross`, built by the `app/lua/lua_cross` make. (Some developers might also use the `spiffsimg` exectable, made in the `tools/spifsimg` subdirectory). Both of these components use the host toolchain (that is the compiler and associated utilities), rather than the Xtensa cross-compiler toolchain, so it is therefore straightforward to make under any environment which provides POSIX runtime support, including WSL, MacOS and Cygwin.

Most Lua developers seem to start with the ESPlorer tool, a 'simple to use' IDE that enables beginning Lua developers to get started. ESPlorer can be slow cumbersome for larger ESP application, and it requires a direct UART connection. So many experienced Lua developers switch to a rapid development cycle where they use a development machine to maintain your master Lua source. Going this route will allow you use your favourite program editor and source control, with one of various techniques for compiling the lua on-host and downloading the compiled code to the ESP:

- If you use a fixed SPIFFS image (I find 128Kb is enough for most of my applications) and are developing on a UART-attached ESP module, then you can also recompile any LC fil LFS image, then rebuild a SPIFFS file system image before loading it onto the ESP using `esptool.py`; if you script this you will find that this cycle takes less than a minute. You can

either embed the LFS.img in the SPIFFS. You can also use the `luac.cross -a` option to build an absolute address format image that you can directly flash into the LFS region within the firmware.
- If you only need to update the Lua components, then you can work over-the-air (OTA). For example see my HTTP_OTA.lua, which pulls a new LFS image from a webservice and reloads it into the LFS region. This only takes seconds, so I often use this in preference to UART-attached loading.
- Another option would be to include the FTP and Telnet modules in the base LFS image and to use telnet and FTP to update your system. (Given that a 64Kb LFS can store thousands of lines of Lua, doing this isn't much of an issue.)

My current practice is to use a small bootstrap `init.lua` file in SPIFFS to connect to WiFi, and also load the `_init` module from LFS to do all of the actual application initialisation. There is a few sec delay whilst connecting to the Wifi, and this delay also acts as a "just in case" when I am developing, as it is enough to allow me to paste a `file.remove('init.lua')` into the UART if my test application is stuck into a panic loop, or set up a different development path for debugging.

Under rare circumstances, for example a power fail during the flashing process, the flash can be left in a part-written state following a `flashreload()`. The Lua RTS start-up sequence will detect this and take the failsafe option of resetting the LFS to empty, and if this happens then the LFS `_init` function will be unavailable. Your `init.lua` should therefore not assume that the LFS contains any modules (such as `_init`), and should contain logic to detect if LFS reset has occurred and if necessary reload the LFS again. Calling `node.flashindex("_init")()` directly will result in a panic loop in these circumstances. Therefore first check that `node.flashindex("_init")` returns a function or protect the call, `pcall(node.flashindex("_init"))`, and decode the error status to validate that initialisation was successful.

No doubt some standard usecase / templates will be developed by the community over the next six months.

A LFS image can be loaded in the LFS store by either during provisioning of the initial firmware image or programmatically at runtime as discussed further in Compiling and Loading LFS Images below. one of two mechanisms:

- The image can be build on the host and then copied into SPIFFS. Calling the `node.flashreload()` API with this filename will load the image, and then schedule a restart to leave the ESP in normal application mode, but with an updated flash block. This sequence is essentially atomic. Once called, and the format of the LFS image has been valiated, then the only exit is the reboot.
- The second option is to build the LFS image using the `-a` option to base it at the c v: release
absolute address of the LFS store for a given firmware image. The LFS can then be flashed to the ESP along with the firmware image.

The LFS store is a fixed size for any given firmware build (configurable by the application developer through `user_config.h`) and is at a build-specific base address within the `ICACHE_FLASH` address space. This is used to store the ROM string table and the set of `Proto` hierarchies corresponding to a list of Lua files in the loaded image.

A separate `node.flashindex()` function creates a new Lua closure based on a module loaded into LFS and more specfically its flash-based prototype; whilst this access function is not transparent at a coding level, this is no different functionally than already having to handle `lua` and `lc` files and the existing range of load functions (`load`,`loadfile`, `loadstring`). Either way, creating a closure on flash-based prototype is *fast* in terms of runtime. (It is basically a single instruction rather than a compile, and it has minimal RAM impact.)

General comments

- **Reboot implementation**. Whilst the application initiated LFS reload might seem an overhead, it typically only adds a few seconds per reboot.
- **Typical Usecase**. The rebuilding of a store is an occasional step in the development cycle. (Say up to 10-20 times a day in a typical intensive development process). Modules and source files under development can also be executed from SPIFFS in `.lua` format. The developer is free to reorder the `package.loaders` and load any SPIFFS files in preference to Flash ones. And if stable code is moved into Flash, then there is little to be gained in storing development Lua code in SPIFFS in `lc` compiled format.
- **Flash caching coherency**. The ESP chipset employs hardware enabled caching of the `ICACHE_FLASH` address space, and writing to the flash does not flush this cache. However, in this restart model, the CPU is always restarted before any updates are read programmatically, so this (lack of) coherence isn't an issue.
- **Failsafe reversion**. Since the entire image is precompiled and validated before loading into LFS, the chances of failure during reload are small. The loader uses the Flash NAND rules to write the flash header flag in two parts: one at start of the load and again at the end. If on reboot, the flag in on incostent state, then the LFS is cleared and disabled until the next reload.