Lua Language

The nodeMCU firmware implements Lua 5.1 over the Espressif SDK for its ESP8266 SoC and the IoT modules based on this.

- The official lua.org **Lua Language specification** gives a terse but complete language specification.
- Its FAQ provides information on Lua availability and licensing issues.
- The **unofficial Lua FAQ** provides a lot of useful Q and A content, and is extremely useful for those learning Lua as a second language.
- The Lua User's Wiki gives useful example source and relevant discussion. In particular, its Lua Learning Lua section is a good place to start learning Lua.
- The best book to learn Lua is *Programming in Lua* by Roberto Ierusalimschy, one of the creators of Lua. It's first edition is available free online . The second edition was aimed at Lua 5.1, but is out of print. The third edition is still in print and available in paperback. It contains a lot more material and clearly identifies Lua 5.1 vs Lua 5.2 differences. **This third edition is widely available for purchase and probably the best value for money**. References of the format [PiL **n.m**] refer to section **n.m** in this edition.
- The Espressif ESP8266 architecture is closed source, but the Espressif SDK itself is continually being updated so the best way to get the documentation for this is to google Espressif IoT SDK Programming Guide or to look at the Espressif downloads forum .
- The **nodeMCU documentation** is available online. However, please remember that the development team are based in China, and English is a second language, so the documentation needs expanding and be could improved with technical proofing.
- As with all Open Source projects the source for the nodeMCU firmware is openly available on the GitHub nodemcu-firmware repository.

How is nodeMCU Lua different to standard Lua?

Whilst the Lua standard distribution includes a host stand-alone Lua interpreter, Lua itself is primarily an *extension language* that makes no assumptions about a “main” program: Lua works embedded in a host application to provide a powerful, light-weight scripting language for use within the application. This host application can then invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code. Through the use of C functions, Lua can be augmented to cope with a wide range of different domains, thus creating customized programming languages sharing a syntactical framework.

The ESP8266 was designed and is fabricated in China by Espressif Systems. Espressif have also developed and released a companion software development kit (SDK) to enable developers to build practical IoT applications for the ESP8266. The SDK is made freely available to developers in the form of binary libraries and SDK documentation. However this is in a *closed format*, with no developer access to the source files, so ESP8266 applications *must* rely solely on the SDK API (and the somewhat Spartan SDK API documentation).

The nodeMCU Lua firmware is an ESP8266 application and must therefore be layered over the ESP8266 SDK. However, the hooks and features of Lua enable it to be seamlessly integrated without loosing any of the standard Lua language features. The firmware has replaced some standard Lua modules that don't align well with the SDK structure with ESP8266-specific versions. For example, the standard `io` and `os` libraries don't work, but have been largely replaced by the nodeMCU `node` and `file` libraries. The `debug` and `math` libraries have also been omitted to reduce the runtime footprint.

NodeMCU Lua is based on eLua, a fully featured implementation of Lua 5.1 that has been optimized for embedded system development and execution to provide a scripting framework that can be used to deliver useful applications within the limited RAM and Flash memory resources of embedded processors such as the ESP8266. One of the main changes introduced in the eLua fork is to use read-only tables and constants wherever practical for library modules. On a typical build this approach reduces the RAM footprint by some 20-25KB and this makes a Lua implementation for the ESP8266 feasible. This technique is called LTR and this is documented in detail in an eLua technical paper: Lua Tiny RAM.

The mains impacts of the ESP8266 SDK and together with its hardware resource limitations are not in the Lua language implementation itself, but in how *application programmers must approach developing and structuring their applications*. As discussed in detail below, the SDK is non-preemptive and event driven. Tasks can be associated with given events by using the SDK API to registering callback functions to the corresponding events. Events are queued internally within the SDK, and it then calls the associated tasks one at a time, with each task returning control to the SDK on completion. *The SDK states that if any tasks run for more than 10 mSec, then services such as Wifi can fail.*

The nodeMCU libraries act as C wrappers around registered Lua callback functions to enable these to be used as SDK tasks. ***You must therefore use an Event-driven programming style in writing your ESP8266 Lua programs***. Most programmers are used to writing in a procedural style where there is a clear single flow of execution, and the program interfaces to operating system services by a set of synchronous API calls to do network I/O, etc. Whilst the logic of each individual task is procedural, this is not how you code up ESP8266 applications.

ESP8266 Specifics

How is coding for the ESP8266 the same as standard Lua?

- This is a fully featured Lua 5.1 implementation so all standard Lua language constructs and data types work.
- The main standard Lua libraries – `core`, `coroutine`, `string` and `table` are implemented.

How is coding for the ESP8266 different to standard Lua?

- The ESP8266 use onchip RAM and offchip Flash memory connected using a dedicated SPI interface. Both of these are *very* limited (when compared to systems than most application programmer use). The SDK and the Lua firmware already use the majority of this resource: the later build versions keep adding useful functionality, and unfortunately at an increased RAM and Flash cost, so depending on the build version and the number of modules installed the runtime can have as little as 17KB RAM and 40KB Flash available at an application level. This Flash memory is formatted an made available as a **SPI Flash File System (SPIFFS)** through the `file`library.
- However, if you choose to use a custom build, for example one which uses integer arithmetic instead of floating point, and which omits libraries that aren't needed for your application, then this can help a lot doubling these available resources. (See Marcel Stör's excellent custom build tool that he discusses in this forum topic). Even so, those developers who are used to dealing in MB or GB of RAM and file systems can easily run out of these resources. Some of the techniques discussed below can go a long way to mitigate this issue.
- Current versions of the ESP8266 run the SDK over the native hardware so there is no underlying operating system to capture errors and to provide graceful failure modes, so system or application errors can easily “PANIC” the system causing it to reboot. Error handling has been kept simple to save on the limited code space, and this exacerbates this tendency. Running out of a system resource such as RAM will invariably cause a messy failure and system reboot.
- There is currently no `debug` library support. So you have to use 1980s-style “binary-chop” to locate errors and use print statement diagnostics though the systems UART interface. (This omission was largely because of the Flash memory footprint of this library, but there is no reason in principle why we couldn't make this library available in the near future as an custom build option).
- The LTR implementation means that you can't easily extend standard libraries as you can in normal Lua, so for example an attempt to define `function table.pack()` will cause a runtime error because you can't write to the global `table`. (Yes, there are standard sand-boxing techniques to achieve the same effect by using metatable based inheritance, but if you try to use this type of approach within a real application, then you will find that you run out of RAM before you implement anything useful.)
- There are standard libraries to provide access to the various hardware options supported by the hardware: WiFi, GPIO, One-wire, I²C, SPI, ADC, PWM, UART, etc.
- The runtime system runs in interactive-mode. In this mode it first executes any `init.lua` script. It then “listens” to the serial port for input Lua chunks, and executes them once syntactically complete. There is no `luac` or batch support, although automated embedded processing is normally achieved by setting up the necessary event triggers in the `init.lua` script.
- The various libraries (`net`, `tmr`, `wifi`, etc.) use the SDK callback mechanism to bind Lua processing to individual events (for example a timer alarm firing). Developers should make full use of these events to keep Lua execution sequences short. *If any individual task takes too long to execute then other queued tasks can time-out and bad things start to happen.*
- Non-Lua processing (e.g. network functions) will usually only take place once the current Lua chunk has completed execution. So any network calls should be viewed at an asynchronous request. A common coding mistake is to assume that they are synchronous, that is if two `socket:send()` are on consecutive lines in a Lua programme, then the first has completed by the time the second is executed. This is wrong. Each `socket:send()` request simply queues the send operation for dispatch. Neither will start to process until the Lua code has return to is calling C function. Stacking up such requests in a single Lua task function burns scarce RAM and can trigger a PANIC. This true for timer, network, and other callbacks. It is even the case for actions such as requesting a system restart, as can be seen by the following example:

```
node.restart(); for i = 1, 20 do print("not quite yet -- ",i); end
```

- You therefore *have* to implement ESP8266 Lua applications using an event driven approach. You have to understand which SDK API requests schedule asynchronous processing, and which define event actions through Lua callbacks. Yes, such an event-driven approach makes it difficult to develop procedurally structured applications, but it is well suited to developing the sorts of application that you will typically want to implement on an IoT device.

So how does the SDK event / tasking system work in Lua?

- The SDK employs an event-driven and task-oriented architecture for programming at an applications level.
- The SDK uses a startup hook `void user_init(void)`, defined by convention in the C module `user_main.c`, which it invokes on boot. The `user_init()` function can be used to do any initialisation required and to call the necessary timer alarms or other SDK API calls to bind and callback routines to implement the tasks needed in response to any system events.
- The API provides a set of functions for declaring application functions (written in C) as callbacks to associate application tasks with specific hardware and timer events. These are non-preemptive at an applications level.
- Whilst the SDK provides a number of interrupt driven device drivers, the hardware architecture severely limits the memory available for these drivers, so writing new device drivers is not a viable options for most developers

- The SDK interfaces internally with hardware and device drivers to queue pending events.
- The registered callback routines are invoked sequentially with the associated C task running to completion uninterrupted.
- In the case of Lua, these C tasks are typically functions within the Lua runtime library code and these typically act as C wrappers around the corresponding developer-provided Lua callback functions. An example here is the Lua `tmr.alarm(id, interval, repeat, callback)` function. The calls a function in the `tmr`library which registers a C function for this alarm using the SDK, and when this C function is called it then invokes the Lua callback.

The nodeMCU firmware simply mirrors this structure at a Lua scripting level:

- A startup module `init.lua` is invoked on boot. This function module can be used to do any initialisation required and to call the necessary timer alarms or libary calls to bind and callback routines to implement the tasks needed in response to any system events.
- The Lua libraries provide a set of functions for declaring application functions (written in Lua) as callbacks (which are stored in the Lua registry) to associate application tasks with specific hardware and timer events. These are non-preemptive at an applications level.
- The Lua libraries work in consort with the SDK to queue pending events and invoke any registered Lua callback routines, which then run to completion uninterrupted.
- Excessively long-running Lua functions can therefore cause other system functions and services to timeout, or allocate memory to buffer queued data, which can then trigger either the watchdog timer or memory exhaustion, both of which will ultimately cause the system to reboot.
- By default, the Lua runtime also 'listens' to UART 0, the serial port, in interactive mode and will execute any Lua commands input through this serial port.

This event-driven approach is very different to a conventional procedural implementation of Lua.

Consider a simple telnet example given in `examples/fragment.lua`:

```
s=net.createServer(net.TCP)
s:listen(23,function(c)
  con_std = c
  function s_output(str)
    if(con_std~=nil) then
      con_std:send(str)
    end
  end
  node.output(s_output, 0)
  c:on("receive",function(c,l) node.input(l) end)
  c:on("disconnection",function(c)
    con_std = nil
    node.output(nil)
  end)
end)
```

This example defines five Lua functions:

Function	Defined in	Parameters	Callback?
Main	Outer module	… (Not used)	
Connection listener	Main	c (connection socket)	
s_output	Connection listener	str	Yes
On Receive	Connection listener	c, l (socket, input)	Yes
On Disconnect	Connection listener	c (socket)	Yes

`s`, `con_std` and `s_output` are global, and no upvalues are used. There is no “correct” order to define these in, but we could reorder this code for clarity (though doing this adds a few extra globals) and define these functions separately one another. However, let us consider how this is executed:

- The outer module is compiled including the four internal functions.
- `Main` is then assigning the created `net.createServer()` to the global `s`. The `connection listener` closure is created and bound to a temporary variable which is then passed to the `socket.listen()` as an argument. The routine then exits returning control to the firmware.

- When another computer connects to port 23, the listener handler retrieves the reference to then connection listener and calls it with the socket parameter. This function then binds the s_output closure to the global `s_output`, and registers this function with the `node.output` hook. Likewise the `on receive` and `on disconnection` are bound to temporary variables which are passed to the respective on handlers. We now have four Lua function registered in the Lua runtime libraries associated with four events. This routine then exits returning control to the firmware.

- When a record is received, the on receive handler within the net library retrieves the reference to the `on receive` Lua function and calls it passing it the record. This routine then passes this to the `node.input()` and exits returning control to the firmware.

- The `node.input` handler polls on an 80 mSec timer alarm. If a compete Lua chunk is available (either via the serial port or node input function), then it executes it and any output is then passed to the `note.output` handler. which calls `s_output` function. Any pending sends are then processed.

- This cycle repeats until the other computer disconnects, and `net` library disconnection handler then calls the Lua `on disconnect` handler. This Lua routine dereferences the connected socket and closes the `node.output` hook and exits returning control to the disconnect handler which garbage collects any associated sockets and registered on handlers.

Whilst this is all going on, The SDK can (and often will) schedule other event tasks in between these Lua executions (e.g. to do the actual TCP stack processing). The longest individual Lua execution in this example is only 18 bytecode instructions (in the main routine).

Understanding how the system executes your code can help you structure it better and improve memory usage. Each event task is established by a callback in an API call in an earlier task.

So what Lua library functions enable the registration of Lua callbacks?

SDK Callbacks include:

Lua Module	Functions which define or remove callbacks
tmr	`alarm(id, interval, repeat, function())`
node	`key(type, function())`, `output(function(str), serial_debug)`
wifi	`startsmart(chan, function())`, `sta.getap(function(table))`
net.server	`sk:listen(port,[ip],function(socket))`
net	`sk:on(event, function(socket, [, data]))`, `sk:send(string, function(sent))`, `sk:dns(domain, function(socket,ip))`
gpio	`trig(pin, type, function(level))`
mqqt	`client:m:on(event, function(conn[, topic, data])`
uart	`uart.on(event, cnt, [function(data)], [run_input])`

So how is context passed between Lua event tasks?

- It is important to understand that any event callback task is associated with a single Lua function. This function is executed from the relevant nodeMCU library C code using a `lua_call()`. Even system initialisation which executes the `dofile("init.lua")` can be treated as a special case of this. Each function can invoke other functions and so on, but it must ultimate return control to the C library code.

- By their very nature Lua `local` variables only exist within the context of an executing Lua function, and so all locals are destroyed between these `lua_call()`actions. *No locals are retained across events*.

- So context can only be passed between event routines by one of three mechanisms:
  - **Globals** are by nature globally accessible. Any global will persist until explicitly dereference by reassigning `nil` to it. Globals can be readily enumerated by a `for k,v in pairs(_G) do` so their use is transparent.

  - The **File system** is a special case of persistent global, so there is no reason in principle why it can't be used to pass context. However the ESP8266 file system uses flash memory and this has a limited write cycle lifetime, so it is best to avoid using the file system to store frequently changing content except as a mechanism of last resort.

  - **Upvalues**. When a function is declared within an outer function, all of the local variables in the outer scope are available to the inner function. Since all functions are stored by reference the scope of the inner function might outlast the scope of the outer function, and the Lua runtime system ensures that any such references persist for the life of any functions that reference it. This standard feature of Lua is known as *closure* and is described in [Pil 6]. Such values are often called*upvalues*. Functions which are global or registered callbacks will persist between event routines, and hence any upvalues referenced by them can be used for passing context.

So how is the Lua Registry used and why is this important?

So all Lua callbacks are called by C wrapper functions that are themselves callback activated by the SDK as a result of a given event. Such C wrapper functions themselves frequently need to store state for passing between calls or to other wrapper C

functions. The Lua registry is simply another Lua table which is used for this purpose, except that it is hidden from direct Lua access. Any content that needs to be saved is created with a unique key. Using a standard Lua table enables standard garbage collection algorithms to operate on its content.

Note that we have identified a number of cases where library code does not correctly clean up Registry content when closing out an action, leading to memory leaks.

Why is it importance to understand how upvalues are implemented when programming for the ESP8266?

Routines directly or indirectly referenced in the globals table, **_G**, or in the Lua Registry may use upvalues. The number of upvalues associated with a given routine is determined by the compiler and a vector is allocated when the closure is bound to hold these references. Each upvalues is classed as open or closed. All upvalues are initially open which means that the upvalue references back to the outer functions's register set. However, upvalues must be able to outlive the scope of the outer routine where they are declared as a local variable. The runtime VM does this by adding extra checks when executing a function return to scan any defined closures within its scope for back references and allocate memory to hold the upvalue and points the upvalue's reference to this. This is known as a closed upvalue.

This processing is a mature part of the Lua 5.x runtime system, and for normal Lua applications development this “behind-the-scenes” magic ensures that upvalues just work as any programmer might expect. Sufficient garbage collector metadata is also stored so that these hidden values will be garbage collected correctly *when properly dereferenced*. However allocating these internal structures is quite expensive in terms of memory, and this hidden overhead is hard to track or to understand. If you are developing a Lua application for a PC where the working RAM for an application is measured in MB, then this isn't really an issue. However, if you are developing an application for the ESP8266 where you might have 20 KB for your program and data, this could prove a killer.

One further complication is that some library functions don't correctly dereference expired callback references and as a result their upvalues may not be correctly garbage collected (though we are tracking this down and hopefully removing this issue). This will all be manifested as a memory leak. So using upvalues can cause more frequent and difficult to diagnose PANICs during testing. So my general recommendation is still to stick to globals for this specific usecase of passing context between event callbacks, and `nil` them when you have done with them.

Can I encapsulate actions such as sending an email in a Lua function?

Think about the implications of these last few answers.

- An action such as composing and sending an email involves a message dialogue with a mail server over TCP. This in turn requires calling multiple API calls to the SDK and your Lua code must return control to the C calling library for this to be scheduled, otherwise these requests will just queue up, you'll run out of RAM and your application will PANIC.

- Hence it is simply ***impossible*** to write a Lua module so that you can do something like:

```
-- prepare message
status = mail.send(to, subject, body)
-- move on to next phase of processing.
```

- But you could code up a event-driven task to do this and pass it a callback to be executed on completion of the mail send, something along the lines of the following. Note that since this involves a lot of asynchronous processing and which therefore won't take place until you've returned control to the calling library C code, you will typically execute this as the last step in a function and therefore this is best done as a tailcall [PiL 6.3].

```
-- prepare message
local ms = require("mail_sender")
return ms.send(to, subject, body, function(status) loadfile("process_next.lua")(status) end)
```

- Building an application on the ESP8266 is a bit like threading pearls onto a necklace. Each pearl is an event task which must be small enough to run within its RAM resources and the string is the variable context that links the pearls together.

When and why should I avoid using tmr.delay()?

If you are used coding in a procedural paradigm then it is understandable that you consider using `tmr.delay()` to time sequence your application. However as discussed in the previous section, with nodeMCU Lua you are coding in an event-driven paradigm.

If you look at the `app/modules/tmr.c` code for this function, then you will see that it executes a low level `ets_delay_us(delay)`. This function isn't part of the nodeMCU code or the SDK; it's actually part of the xtensa-lx106 boot ROM, and is a simple timing loop which polls against the internal CPU clock. It does this with interrupts disabled, because if they are enabled then there is no guarantee that the delay will be as requested.

`tmr.delay()` is really intended to be used where you need to have more precise timing control on an external hardware I/O (e.g. lifting a GPIO pin high for 20 µSec). It will achieve no functional purpose in pretty much every other usecase, as any other system code-based activity will be blocked from execution; at worst it will break your application and create hard-to-diagnose timeout errors.

The latest SDK includes a caution that if any (callback) task runs for more than 10 mSec, then the Wifi and TCP stacks might fail, so if you want a delay of more than 8 mSec or so, then *using `tmr.delay()` is the wrong approach*. You should be using a timer alarm or another library callback, to allow the other processing to take place. As the nodeMCU documentation correctly advises (translating Chinese English into English): *`tmr.delay()` will make the CPU work in non-interrupt mode, so other instructions and interrupts will be blocked. Take care in using this function.*

How do I avoid a PANIC loop in init.lua?

Most of us have fallen into the trap of creating an `init.lua` that has a bug in it, which then causes the system to reboot and hence gets stuck in a reboot loop. If you haven't then you probably will do so at least once.

- When this happens, the only robust solution is to reflash the firmware.
- The simplest way to avoid having to do this is to keep the `init.lua` as simple as possible – say configure the wifi and then start your app using a one-time`tmr.alarm()` after a 2-3 sec delay. This delay is long enough to issue a `file.remove("init.lua")` through the serial port and recover control that way.
- Also it is always best to test any new `init.lua` by creating it as `init_test.lua`, say, and manually issuing a `dofile("init_test.lua")` through the serial port, and then only rename it when you are certain it is working as you require.

Techniques for Reducing RAM and SPIFFS footprint

How do I minimise the footprint of an application?

- Perhaps the simplest aspect of reducing the footprint of an application is to get its scope correct. The ESP8266 is an IoT device and not a general purpose system. It is typically used to attach real-world monitors, controls, etc. to an intranet and is therefore designed to implement functions that have limited scope. We commonly come across developers who are trying to treat the ESP8266 as a general purpose device and can't understand why their application can't run.
- The simplest and safest way to use IoT devices is to control them through a dedicated general purpose system on the same network. This could be a low cost system such as a RaspberryPi (RPi) server, running your custom code or an open source home automation (HA) application. Such systems have orders of magnitude more capacity than the ESP8266, for example the RPi has 2GB RAM and its SD card can be up to 32GB in capacity, and it can support the full range of USB-attached disk drives and other devices. It also runs a fully featured Linux OS, and has a rich selection of applications pre configured for it. There are plenty of alternative systems available in this under $50 price range, as well as proprietary HA systems which can cost 10-50 times more.
- Using a tiered approach where all user access to the ESP8266 is passed through a controlling server means that the end-user interface (or smartphone connector), together with all of the associated validation and security can be implemented on a system designed to have the capacity to do this. This means that you can limit the scope of your ESP8266 application to a limited set of functions being sent to or responding to requests from this system.
- *If you are trying to implement a user-interface or HTTP webserver in your ESP8266 then you are really abusing its intended purpose. When it comes to scoping your ESP8266 applications, the adage **K**eep **I**t **S**imple **S**tupid truly applies.*

How do I minimise the footprint of an application on the file system

- It is possible to write Lua code in a very compact format which is very dense in terms of functionality per KB of source code.
- However if you do this then you will also find it extremely difficult to debug or maintain your application.
- A good compromise is to use a tool such as LuaSrcDiet, which you can use to compact production code for downloading to the ESP8266:
  - Keep a master repository of your code on your PC or a cloud-based versioning repository such as GitHub
  - Lay it out and comment it for ease of maintenance and debugging
  - Use a package such as Esplorer to download modules that you are debugging and to test them.
  - Once the code is tested and stable, then compress it using LuaSrcDiet before downloading to the ESP8266. Doing this will reduce the code footprint on the SPIFFS by 2-3x.
- Consider using `node.compile()` to pre-compile any production code. This removes the debug information from the compiled code reducing its size by roughly 40%. (However this is still perhaps 1.5-2x larger than a LuaSrcDiet-compressed source format, so if SPIFFS is tight then you might consider leaving less frequently run modules in Lua format. If you do a compilation, then you should consider removing the Lua source copy from file system as there's little point in keeping both on the ESP8266.

How do I minimise the footprint of running application?

- The Lua Garbage collector is very aggressive at scanning and recovering dead resources. It uses an incremental mark-and-sweep strategy which means that any data which is not ultimately referenced back to the Globals table, the Lua registry or in-scope local variables in the current Lua code will be collected.
- Setting any variable to `nil` dereferences the previous context of that variable. (Note that reference-based variables such as tables, strings and functions can have multiple variables referencing the same object, but once the last reference has been set to `nil`, the collector will recover the storage.
- Unlike other compile-on-load languages such as PHP, Lua compiled code is treated the same way as any other variable type when it comes to garbage collection and can be collected when fully dereferenced, so that the code-space can be

reused.

- Lua execution is intrinsically divided into separate event tasks with each bound to a Lua callback. This, when coupled with the strong dispose on dereference feature, means that it is very easy to structure your application using an classic technique which dates back to the 1950s known as Overlays.

- Various approaches can be use to implement this. One is described by DP Whittaker in his Massive memory optimization: flash functions topic. Another is to use*volatile modules*. There are standard Lua templates for creating modules, but the `require()` library function creates a reference for the loaded module in the`package.loaded` table, and this reference prevents the module from being garbage collected. To make a module volatile, you should remove this reference to the loaded module by setting its corresponding entry in `package.loaded` to `nil`. You can't do this in the outermost level of the module (since the reference is only created once execution has returned from the module code), but you can do it in any module function, and typically an initialisation function for the module, as in the following example:

```
-- . . .
local s=net.createServer(net.TCP)
s:listen(80,function(c) require("connector").init(c) end)
```

- **`connector.lua`** would be a standard module pattern except that the `M.init()` routine must include the lines

```
local M, module = {}, ...
-- . . .
function M.init(csocket)
  package.loaded[module]=nil
  -- . . .
end
-- . . .
return M
```

- This approach ensures that the module can be fully dereferenced on completion. OK, in this case, this also means that the module has to be reloaded on each TCP connection to port 80; however, loading a compiled module from SPIFFS only takes a few mSec, so surely this is an acceptable overhead if it enables you to break down your application into RAM-sized chunks. Note that `require()` will automatically search for `connector.lc` followed by `connector.lua`, so the code will work for both source and compiled variants.

- Whilst the general practice is for a module to return a table, [PiL 15.1] suggests that it is sometimes appropriate to return a single function instead as this avoids the memory overhead of an additional table. This pattern would look as follows:

```
-- . . .
local s=net.createServer(net.TCP)
s:listen(80,function(c) require("connector")(c) end)
local module = ...  -- this is a situation where using an upvalue is essential!
return function (csocket)
  package.loaded[module]=nil
  module = nil
  -- . . .
end
```

- Also note that you should ***not*** normally code this up listener call as the following because the RAM now has to accommodate both the module which creates the server *and* the connector logic.

```
-- . . .
local s=net.createServer(net.TCP)
local connector = require("connector") -- don't do this unless you've got the RAM available!
s:listen(80,connector)
```

How do I reduce the size of my compiled code?

Note that there are two methods of saving compiled Lua to SPIFFS:

1. The first is to use `node.compile()` on the `.lua` source file, which generates the equivalent bytecode `.lc` file. This approach strips out all the debug line and variable information.

2. The second is to use `loadfile()` to load the source file into memory, followed by `string.dump()` to convert it in-memory to a serialised load format which can then be written back to a `.lc` file. This approach creates a bytecode file which retains the debug information.

The memory footprint of the bytecode created by method (2) is the same as when executing source files directly, but the footprint of bytecode created by method (1) is typically **60% of this size**, because the debug information is almost as large as the code itself. So using `.lc` files generated by `node.compile()` considerably reduces code size in memory – albeit with the downside that any runtime errors are extremely limited.

In general consider method (1) if you have stable production code that you want to run in as low a RAM footprint as possible. Yes, method (2) can be used if you are still debugging, but you will probably be changing this code quite frequently, so it is easier to stick with `.lua` files for code that you are still developing.

Note that if you use `require("XXX")` to load your code then this will automatically search for `XXX.lc` then `XXX.lua` so you don't

need to include the conditional logic to load the bytecode version if it exists, falling back to the source version otherwise.

How do I get a feel for how much memory my functions use?

- You should get an overall understanding of the VM model if you want to make good use of the limited resources available to Lua applications. An essential reference here is A No Frills Introduction to Lua 5.1 VM Instructions . This explain how the code generator works, how much memory overhead is involved with each table, function, string etc..

- You can't easily get a bytecode listing of your ESP8266 code; however there are two broad options for doing this:
  - **Generate a bytecode listing on your development PC**. The Lua 5.1 code generator is basically the same on the PC and on the ESP8266, so whilst it isn't identical, using the standard Lua batch compiler `luac` against your source on your PC with the `-l -s` option will give you a good idea of what your code will generate. The main difference between these two variants is the size_t for ESP8266 is 4 bytes rather than the 8 bytes size_t found on modern 64bit development PCs; and the eLua variants generate different access references for ROM data types. If you want to see what the `string.dump()` version generates then drop the `-s` option to retain the debug information.

  - **Upload your `.lc` files to the PC and disassemble then there**. There are a number of Lua code disassemblers which can list off the compiled code that you application modules will generate, `if` you have a script to upload files from your ESP8266 to your development PC. I use ChunkSpy which can be downloadedhere , but you will need to apply the following patch so that ChunkSpy understands eLua data types:

```
--- a/ChunkSpy-0.9.8/5.1/ChunkSpy.lua   2015-05-04 12:39:01.267975498 +0100
+++ b/ChunkSpy-0.9.8/5.1/ChunkSpy.lua   2015-05-04 12:35:59.623983095 +0100
@@ -2193,6 +2193,9 @@
           config.AUTO_DETECT = true
         elseif a == "--brief" then
           config.DISPLAY_BRIEF = true
+        elseif a == "--elua" then
+          config.LUA_TNUMBER = 5
+          config.LUA_TSTRING = 6
         elseif a == "--interact" then
           perform = ChunkSpy_Interact
```

- Your other great friend is to use `node.heap()` regularly through your code.

- Use these tools and play with coding approaches to see how many instructions each typical line of code takes in your coding style. The Lua Wiki gives some generaloptimisation tips, but in general just remember that these focus on optimising for execution speed and you will be interested mainly in optimising for code and variable space as these are what consumes precious RAM.

What is the cost of using functions?

Consider the output of `dofile("test1a.lua")` on the following code compared to the equivalent where the function `pnh()` is removed and the extra `print(heap())`statement is placed inline:

```lua
-- test1b.lua
collectgarbage()
local heap = node.heap
print(heap())
local function pnh() print(heap()) end
pnh()
print(heap())
```

Heap Value	Function Call	Inline
1	20712	21064
2	20624	21024
3	20576	21024