OPEN AT® DYNAMIC PROGRAMMING WITH LUA

REFERENCE MANUAL

wavecom
Make it wireless

Operating Systems | Integrated Development Environments | Plug-Ins | Wireless CPUs | Services

1 Introduction 5

1.1 WIP Lua: a dynamic language on an embedded platform 5

1.2 Applications 6

1.3 Learning Lua 6
1.3.1 Resources 6
1.3.2 Minimal knowledge 7

1.4 Going further 8

1.5 Manual organization 9

2 Getting started 9

3 Lua API 12

3.1 Limitations of Lua features 12

3.2 Scheduling 13

3.3 WIP 20
3.3.1 bearers 20
3.3.2 channels 23
3.3.3 tcp 28
3.3.4 udp 29
3.3.5 ping 30
3.3.6 fcm 30

3.4 Flash objects 30

3.5 at 31

3.6 sms 34

3.7 shell 35

3.8 proc 37

3.9 misc 37

4 C API 39

4.1 Lua state handling 39

4.2 WIP interface from C 41

4.3 Optimized memory pools 43

5 Samples showcase 44

5.1 HTTP server 44

5.2 FTP server 44

5.3 XML encoder and decoder 44

5.4 Multi-protocol application 44

6 Other tools 45

6.1 off-line compilation 45

Appendix: Setting up Windows for TCP/IP over PPP Client 47

6.2 Configure the serial port 47

6.3 Add the devkit as a modem 48

6.4 Create the connection 50

7 Appendix: anatomy of a Lua application 53

Document Information

Level	Date	History of the evolution	Writer
001	27/08/2007	Creation	FFT(Fabien Fleutot)

1 Introduction

1.1 WIP Lua: a dynamic language on an embedded platform

This library offers you the ability to program your wireless CPU in an arbitrary mix of C and of the dynamic language Lua. The Lua port is not merely a toy to write simplistic programs that would fit on a cheap microcontroller; it features, among others:

- a carefully designed API, including complete bindings to AT, ADL Open AT and WIP (TCP/IP) features,
- multithreading (dynamically created tasks, in arbitrary numbers),
- synchronization on ADL, WIP, AT, Lua and user-defined events,
- automatic garbage collection,
- object-oriented programming,
- runtime (and optionally over the air) diagnostic, debugging and inspection,
- advanced strings and data structures handling,
- easy integration with your C code...

The kind of Lua samples provided (multithreaded FTP and HTTP+Ajax servers in a couple hundreds of lines each, advanced data monitoring and reporting, etc.) clearly demonstrates how empowering Lua is for wireless CPU programming.

The approach adopted is not to replace C: C is here to stay in embedded software, as soon as you need realtime performances or a tight control of your resources; so the right approach is to provide a high-level language, which does well everything that would be tedious in C, and interfaces seamlessly with C for those domains where C shines. That way, you use the right tool for the right job, rather than a single tool which makes everything almost possible but quite cumbersome.

However, by using Lua, you'll probably find yourself writing 95% of your code in Lua, and the remaining 5% in C.

Finally, Lua for wireless CPUs intends not only to let you write more powerful programs, but to streamline the whole development cycle. Development is accelerated:

- By the availability of an interactive shell, over telnet or UARTs, which lets you examine/extend/modify your programs and your data, possibly while they're running.
- By automation and standard protocols support (everything over TCP/IP, sources fetched through FTP and compiled automatically in a matter of seconds, telnet that goes over all bearers including GPRS).
- By live and remote introspection capabilities. Eventually, this experience will be further simplified by a closer integration with Eclipse (think remote debugging over GPRS, for instance).
- By maximizing the flexibility of code: code can be easily downloaded from the network, incorporated in a C application, stored/updated/retrieved/ deleted in/from flash objects.

1.2 Applications

Besides the dramatic productivity boost, due to the language features and the simplified development cycle, embedding the Lua VM and compiler in your application enables many interesting capabilities:

- Remote diagnostic and troubleshooting, if the shell is enabled. Beware that this might require some additional security measures when not working in a private APN/network.
- Improved remote device management: Lua functions can be updated one-by-one, on the fly, without rebooting. Modifications can be freely and safely tested in RAM before being committed to flash with a simple call to save(). Even Lua functions hard-coded in the C application can be overridden by an additional function modification in a flash object. Thus you get the best fine-grain control, network bandwidth economy and upgrade safety you could dream of. With Lua running on servers as well as terminal devices, you get an homogeneous environment in which developing complex, distributed applications.
- Improved configurability. Configuration settings easily consist of arbitrarily complex tables, with optional and default values, dynamically resizable tables, etc.: these are possible to do in C, but at prohibitive development cost, and thus rarely implemented in practice. More dramatically, configurations can easily embed arbitrary functions, e.g. to specify a reporting policy, a "door-knocking" authentication policy, etc.

1.3 Learning Lua

One of Lua's distinguishing features is its ease of learning. You'll master its mainstream features in an hour if you know a dynamic language such as Perl, Python, Ruby, PHP or Javascript, and coming from a pure Java or C background should take you only marginally longer. If you're interested by advanced programming techniques (function closures, coroutines, meta object protocol, weak referencing, code reification...), they are provided by Lua, and you can learn and use them although you don't have to.

1.3.1 Resources

Learning Lua is easy, and many resources exist online and offline.

- The best resource is probably Roberto Ierusalismschy's "*Programming in Lua*"[1].

Online resources include:

[1] "Programming in Lua, 2nd edition"
Published by Lua.org, March 2006
ISBN 85-903798-2-5 Paperback, 328 pages, 1.8 x 24.6 x 18.9cm
Distributed by Ingram and Baker & Taylor.
Also available in German and in Korean.

- The first edition of "Programming in Lua" is freely available online: http://www.lua.org/pil
- The reference manual: http://www.lua.org/manual/5.1
- At-a-glance cheat sheets of the language syntax, features and standard libraries: http://lua-users.org/wiki/LuaShortReference
- A very active user community: http://lua-users.org/wiki. Check their comprehensive list of tutorials and sample code.
- The users' mailing list: http://www.lua.org/lua-l.html, and the IRC channel `#lua` on `irc.freenode.net`

Testimonials

If you would like a bit of advocacy, you can have a look at:

- User feedbacks: http://www.lua.org/quotes.html
- A list of projects which advertize their use of Lua: http://www.lua.org/uses.html
- Mark Hamburg's feedback on writing a big application in Lua: Adobe Photoshop Lightroom, written 40% in Lua, 60% in C and C++. Interview here: http://since1968.com/article/190/mark-hamburg-interview-adobe-photoshop-lightroom-part-2-of-2. Insightful presentation slides at http://www.lua.org/wshop05/Hamburg.pdf.

1.3.2 Minimal knowledge

To write an application in Lua, the minimum you need to know is listed below.

- syntax of control structures: for loops, do/while, repeat/until, if/then/else, function declaration. All these constructions' syntax and semantics are perfectly straightforward, and similar to those found in all procedural languages. They're all summed up in a single page of the short reference referenced above.

- Variable assignments, global and local variable declarations: a variable declared as local only exists wihtin the structure where it's declared; non declared variables are global by default. Optionnally, global variables declaration might be made mandatory, which helps detect some bugs. There is no type declarations, as values are typed dynamically; runtime type-checking is however available, to make your APIs safer.

- Data: simple types include booleans, numbers, strings, functions, threads, userdata (i.e. C data reified in Lua, e.g. WIP channels and bearers). Composite data rely on the amazingly simple yet powerful table datatype, which can be used as a list, a queue, a FIFO, a record (i.e. a C "struct"), an object, a hashtable... Tables will effectively address all the needs you might have for advanced data structures.
  It should also be noted that:

  - strings are pooled: there always exists a single memory copy of a given string. This means that equality test for strings is as cheap as a pointer comparison. The counterpart is that strings are unmutable: instead of modifying a string, the system creates modified copies of it. To avoid impacting the performances in time

and memory, OpenAT data streams are therefore represented by dedicated buffers data structures (themselves implemented with tables), and are only converted to strings on demand.

- Functions are first order values: this means that everything you could do with a string or an number, you can also do it with a function: passing it as a parameter, creating it and returning it as a result, putting it in global or local variables, storing it in flash, in tables, as a C constant, send it through the network...

- Numbers in OpenAT Lua are integers: for the sake of simplicity, Lua works with a single number type, and working with floats is not a good idea when you don't have an FPU[2]. You can still reintroduce IEEE floats as if you need them.

- Knowing the main APIs which control OpenAT through Lua: Access to task scheduling, flash, TCP/IP channels, V24 channels, GPIO, AT commands...

- Interfacing C and Lua. Lua and C user code communicate through a pseudo-stack API: from C, you can push and pop values from the stack, with function such as `lua_pushinteger(L, number)` or `lua_tostring(L, stack_idx)`. From Lua, you can call any C function which takes parameters from a stack and pushes back results on it. Most lua primitive operations are also accessible from C through an extensive dedicated API (although it's often best to do everything you can in Lua).
  The C functions you'll be most interested with are those which translate data between Lua and C. Lua→C translation functions are generally called `luaL_checkxxx( L, idx)`; C→Lua ones are called `lua_toxxx( L, x)`; keep a bookmark on them in the reference manual.

1.4 Going further

If you want to go further, either out of curiosity or because your application is complex enough to leverage advanced programming technics, you might wish to study the following advanced subject:

- Coroutines: the collaborative threading system of Lua. Our platform gives you an easy to use interface which will cater for most of your needs, but you can do even more with the raw coroutines.

- Metatables: you can completely bypass Lua's mechanism from within Lua. We use this, for instance, to implement the "magic" `flash` table, which saves its content automatically in flash, yet is handled like a regular table by the user. It also easily allows object oriented programming, creation of advanced user types, personalized printing functions...

[2] Many portages of Lua, at least on PC, work exclusively with IEEE double floats.

- Full function closures: being able to declare local functions, which capture all of the local variables accessible at their declaration point, open a lot of possibilities, only accessible to very high level languages.

- Strings pattern matching (a.k.a. regular expressions) will save you a lot of time when parsing typical ASCII-based network protocols.

- Interactive debugging: the standard debug library has amazing capabilities for inspecting the stacks, the memory, and monitoring the virtual machine, far beyond any static language and most dynamic ones. It is quite an astonishing diagnostic tool, including for maintenance of already deployed modules.

1.5 Manual organization

This reference manual is organized as follows:

- Getting started: how to setup your computer for development.
- Lua API: all the functions and modules available from Lua to drive the WMP. This API only defines in what way Lua on WMP differs from the original Lua distribution, it doesn't substitute itself to the official manual.
- C API: as for Lua API, it doesn't substitute itself to the official manual. It mainly describes interfacing with an ADL application/library, and optimized memory management.
- Application samples: some demonstrations of what can be easily achieved on Wavecom hardware with Lua.

2 Getting started

We assume that WIP Lua is running on your devkit, and that a PPP connection is established between the computer and the devkit. For details about setting up a PPP client on serial port under Microsoft Windows XP, see in appendix.

Required tools

WIP Lua only relies on standard tools: you'll need an FTP server (filezilla server, WFTPD, proftp...) accepting anonymous downloads, and a Telnet client (under MS-Windows, we strongly recommend Putty, which can be downloaded at http://www.chiark.greenend.org.uk/~sgtatham/putty/). Finally, you'll need a text editor. Although Notepad would theoretically be enough, we advise you to choose a decent one, that supports at least syntax highlight for Lua: all major editors (Eclipse, Emacs, UltraEdit, TextMate...) do, possibly through a plug-in. There is also a wealth of specialized Lua editors, which can be found in http://lua-users.org/wiki/LuaAddons, under the section "Development environments".

First contact

You can check that the devkit is properly connected by pinging it. From now we'll assume that the devkit is at address 192.168.1.4, and the PC at 192.168.1.5: this is the default config of the sample. If it doesn't suit you, you can change it by typing the following AT commands:

```
AT+LUA="LOCAL_ADDR='192.168.131.1'; save 'LOCAL_ADDR'"
OK
AT+LUA="PEER_ADDR='192.168.131.2'; save 'PEER_ADDR'"
OK
```

Interactive Shell Now, connect with Putty to your devkit:

```
Lua interactive shell
$ _
```

Interactive commands You can type your first program:

```
$ print "Hello world"
Hello world
$ _
```

Live inspection/update You can inspect and change your system's settings:

```
$ = proc.channels
= {
  [403572792] = [channel TCPSERVER 0x180e0838 ready],
  [403503416] = [channel TCPCLIENT 0x180cf938 ready] }
$ sh_chan = proc.channels[403503416] -- this is the socket serving the telnet shell
$ = sh_chan.ttl
= 64
$ sh_chan.ttl=255 -- change the socket's Time-To-Live on the fly
$ = sh_chan.ttl
= 255 -- value changed dynamically!
$ _
```

Download & run from FTP Finally, you can run programs. Suppose that your FTP server contains the following program in its anonymous root directory:

```
-- File 'first_program.lua'
for i=1,5 do
  print ("three times", i, "=", 3*i)
end
```

To download, compile and run it, simply type in the shell:

```
$ l 'first_program'
Loading first_program.lua
Loaded. Eval...
three times     1       =       3
three times     2       =       6
three times     3       =       9
three times     4       =       12
three times     5       =       15
Done. Lua VM uses 63330 bytes.
$ _
```

If you want to modify it, edit the file, save it, and reload with `l()`. Notice that `l()` remembers its last argument even across reboots, so you can reboot your devkit between two runs if you wish:

```
$ l() -- In the file first_program.lua, we've changed "for i=1,5" into "for i=1,3"
Loading first_program.lua
Loaded. Eval...
three times     1       =       3
three times     2       =       6
three times     3       =       9
Done. Lua VM uses 63450 bytes.
$ _
```

3 Lua API

3.1 Limitations of Lua features

In order to make Lua run on Wavecom's platform, we had to put a couple of limitations on the implementation:

- No floating point numbers: Lua works with a single number type, which by default is the IEEE double float. Due to the kind of applications typically developed on WMP, and the absence of an FPU, we chose the integral variant of Lua compiler and virtual machine, which use 32 bits signed integers as numbers.
- Some default libraries aren't available: math (relies on floats), io (requires a filesystem), OS (requires an underlying POSIX OS). The base functions `loadfile()` and `dofile()` are also currently disabled, due to lack of a filesystem.
- Coroutines (a.k.a. green threads) are used to implement multitasking; using raw coroutines is therefore tricky, and requires special care, to avoid crashing the scheduler. A section will be dedicated to the subject. Anyway, many realistic use cases for raw coroutines are superseded by the scheduling and signalling API.

All other standard libraries, base functions and features are fully supported.

3.2 Scheduling

Lua port on WCPU has its own collaborative multitasking system, which runs in a single OS task. Tasks are created and destroyed dynamically, limited in number only by available memory (about 1.5KB RAM per thread). Functions run(), signal() and wait() are at the core of multitasking in Lua.

All Lua collaborative threads run in a single user ADL OS task. They give back the hand either by calling `wait()`, but some other functions might cause the hand to be given back as well (most probably by calling `wait()` in their implementation). These functions are marked with a ASYNC label in the manual.

A remark reserved to expert Lua programmers: you should be warned that the system makes heavy use of Lua coroutines: be careful if you must use coroutines on your own, all functions marked with ASYNC might cause a `coroutine.yield()`. Take this into account, and if you can (i.e. in most cases), write your stuff in terms of synchronized tasks rather than coroutines.

run(x [,args])

Schedule a task for running. x can be:

- a sleeping thread, which is then reactivated
- a function, which is turned into a thread and scheduled for running. If some extra parameters are given, they are passed to f.

If a scheduler is already running, run() simply returns the thread created or woken up. If no scheduler is running (which probably means that run() has been launched from C, typically in an ADL/WIP callback), then the scheduler is woken up until no thread can run anymore (all dead or waiting for a signal).

Beware that Lua's multitasking is collaborative: it's up to the tasks to give up the hand, either by calling wait() or a blocking function (which in turns eventually calls wait()). The advantages are economy of resources, and very limited risks of race conditions and deadlocks; the corresponding drawbacks are that a single task can freeze the whole Lua VM, and in some (pretty rare) cases, the user has to think about explicitly giving back the hand to the scheduler, so that other tasks have a chance to run.

FIXME the scheduler should be modified to yield regularly, to make sure the watchdog isn't triggered by expensive computations.

Examples

Accumulate all lines received on a socket in the background (i.e. the shell remains available, and other tasks can run concurrently as well)

```
accumulator = { }
function accumulate_from_socket (addr, port)
  local x = wip.tcp_client(addr, port)
  while true do
    local line = x:read "*l"
    table.insert (accumulator, line)
  end
end
run(accumulate_socket, 192.168.1.4, 2007)
```

You can check the accumulator's content at any time, while filling goes on in the background:

```
$ = accumulator
= { "foo\r\n", "bar\r\n" }
$ = accumulator -- after having waited a while, and data arrived on the socket
= { "foo\r\n", "bar\r\n", "gnat\r\n" }
$ _
```

signal(emitter, event [,args])

Signal event event on behalf of emitter. emitter can be any object, event must be a string. If some extra args are given, they're passed to the hooks and the threads waiting for that signal.

Signals can be emitted from C. They are used to advertise AT, ADL and WIP events to Lua, thus allowing to synchronize Lua application on relevant hardware events, e.g. waiting for a channel to close, for an AT command to terminate etc.

Examples: This only makes sense with a task waiting for the event, or a callback being attached to it. see wait().

wait(emitters, events) ASYNC

wait(tenths_of_seconds)

wait()

Put the currently running thread to sleep until one of the objects in table emitters receives one of the events in table events. In cases where only one emitter and/or one event is to be waited for, it can be put out of tables, i.e. `wait(channel, "close")` is the same as `wait({channel}, {"close"})`.

Return the emitter, the event, and any additional parameter associated to the event by the `signal()` call.

Asleep tasks can be manually managed and killed from table `proc.tasks.waiting`.

When called without any argument, `wait()` simply puts back the current tasks at the bottom of the list of schedulable tasks: when a tasks might keep the hand for a long time, periodically calling wait() that way gives a chance to other tasks to run as well, thus keeping a multithread feeling. In practice, however, many functions such as TCP or AT ones implicitly give back the hand, when waiting of ADL/WIP events: most reasonable applications tend to keep collaborative multitasking running smoothly.

Moreover, `wait()` behaves in a special way if one of the events is a positive number `n` rather than a string: it is a timer event that will happen `n` 10th of seconds later (at least: we're not realtime, remember): since `wait()` returns as soon as one of the events happens, it will return after `n` 10th of seconds at the latest. When `wait()` returns because of a timer, it returns an event of the form "`@xxx.d`" where `xxx` is the epoch of the timer expiration (i.e. the date, as a number of seconds since 1/1/1970 midnight), and `d` is a number of 10th of seconds, added to the epoch's number of seconds.

As a special case, if you only want to wait of a certain delay `n` (still in 10th of seconds), you don't have to precise an emitter at all: `wait(n)` will work.

Examples

Write "`plop`" on the console every 5 seconds, in a background tasks:

```
run (function() while true do print "plop"; wait(50) end end)
```

Read on a socket and write whatever is received to the console. If nothing is received during a 30 second period, or a socket event occurs, or the peer socket closes the connection, we stop and close the socket.

```
x=wip.tcp_client(PEER_ADDR, 2007)
while true do
  local em, ev = wait(x, {'read', 'peer_close', 'error', 300})
  assert( em==x)
  if ev ~= 'read' then break end
  local msg = x:read()
  print(msg)
end
x:close()
```

Let's do it again, in a more contrived way: one task counter will emit signal "foo"."bar" every 5 seconds, and two other tasks writer1 and writer2 will write "plic" and "plop" everytime it sees that signal "foo"."bar". Moreover, an additional parameter i is associated to the event and passed to the PLIC and PLOP generators:

```
function counter() -- generate "foo"."bar" every 5 seconds for 5000 seconds
   for i = 1, 1000 do
      signal("foo", "bar", i)
      wait(50)
   end
end

function writer1()
   while true do
      local emitter, event, i = wait("foo", "bar")
      printf ("PLOP number %i", i)
   end
end

function writer2()
   while true do
      local emitter, event, i = wait("foo", "bar")
      printf ("PLIC number %i", i)
   end
end

run(counter); run(writer1); run(writer2) -- run the 3 functions in parallel
```

sighook(emitters, events, hook)

Attach a `hook` function to be triggered when one of `events` is signalled by one of `emitters`. As with `wait()`, when a single emitter/event is monitored, it can stay out of a table.

The function `hook` receives as parameters the emitter, the event and any extra arguments passed to the `signal()` call which triggered it. If `hook()` returns `false` or `nil`, it is detached from the event. If it returns anything else, it stays attached and will be triggered again next time a suitable emitter/event pair is signalled. It's good practice, when a hook requests to stay attached, to make it return the string "`again`".

In an interactive shell, a practical way to kill a hook which never detaches itself is to empty the suitable entry(ies) in `proc.tasks.waiting`. Such entries can be recognized by the fact that they bear a hook field.

IMPORTANT WARNING: signal hooks are supposed to be short synchronous reactions. They cannot run any asynchronous (blocking) function. If a function in a signal tries to access to scheduling, it will fail. If you want to use an asynchronous function in a signal hook, put it in a `run(function()…end)` call. In that case, don't expect the function inside `run(…)` to be executed synchronously.

Examples

Here is an idiomatic way to attach a callback to a channel which will, upon error or shutdown, close it and set it to nil automatically:

```
c = wip.tcp_client('www.wavecom.com', 80)
sighook( c, {'error','peer_close'},
         function (emitter, event) c:close(); c=nil end)
```

This one will print a message every time the SIM rack is opened, by catching AT events "+WIND: 14"

```
local function watch_rack(event, emitter, arg)
   if arg=='14' then print 'Rack opened!' end
   return 'again' -- don't detach the hook
end
sighook('at','WIND',watch_rack)
```

kill(task)

Kill a task, whether it's waiting for an event or ready for scheduling. The preferred way to get the task id is to gather it when it's returned from the first call to `run()`.

Example

In this shell session, we create a function that regularly prints a message in the background, then stops it by killing the task.

```
$ task = run(function(x) while true do print 'plop!'; wait(20) end end)
$ plop!
plop!
plop!
plop!
kill(task)
$ _
```

Notice that a killed task still emits a "`die`" signal:

```
$ do
+   task = run(function(x) while true do print 'plop!'; wait(20) end end)
+   sighook(task, "die", function() print "AAaaaarghHh!" end)
+ end
$ plop!
plop!
plop!
plop!
plop!
kill(task)
AAaaaarghHh!
$ _
```

3.3 WIP

3.3.1 bearers

Bearers are started with `wip.bearer_client()` and `wip.bearer_server()`. These function are synchronous: they only return when the bearer is up and running. If you don't want to wait for bearer establishement, use `run()` to put the bearer config in a background task. Since some options need to be passed before the connection is started, the constructors accept a table of options as a second parameter. Once the bearer is ready, you can read and write the options as regular fields (provided they're readable/writable). In addition to usual options as listed below, GSM/GPRS bearers accept a "pin" pseudo-option that lets you set the PIN number during the config.

FIXME: GSM connection not tested.

`wip.bearer_client(name, opts)`

Create a bearer. The name has to be one of **FIXME**. opts is an option_name → option value association table: each of these options will be set before the bearer is started. The list of available options is given below, and their detailed meaning can be found in the WIP user's guide.

`'GPRS'` bearer has, for convenience, an additional pseudo-option `'pin'`: when set, it enters the PIN code and waits for full SIM init before starting actual GPRS bearer registration.

Once created, the bearers are registered in the table proc.bearers. They're referenced by name for easy user retrieval (e.g. the GPRS bearer, when running, is always registered as proc.bearers.GPRS), and by numeric key for easy programmatic retrieval.

Examples

The following program will allow you to easily set the GPRS link up by calling `gprs()`, and to tune the config options by modifying `grps_config`:

```
function gprs()
   wip.bearer_client('GPRS', gprs_config)
end

gprs_config = { pin      = 1234;
                apn      = 'websfr';
                login    = 'foo';
                password = 'bar' }

save('gprs', 'gprs_config') -- commit to flash
```

wip.bearer_server(name, opts)

Works the same as wip.bearer_client(), except that the accepted names are FIXME.

BEARER:close()

Destroy the bearer, and dereferences it from proc.bearers.

Bearer options

Bearer options can be accessed as regular fields of a table. Here is the correspondence between WIP options and their Lua counterpart:

WIP option name	Lua field name	Lua type
WIP_BOPT_GPRS_APN	apn or gprs_apn	STRING
WIP_BOPT_GPRS_CID	cid or gprs_cid	NUMBER
WIP_BOPT_GPRS_DATACOMP	datacomp or gprs_datacomp	BOOL
WIP_BOPT_DIAL_MSNULLMODEM	dial_msnullmodem	BOOL
WIP_BOPT_DIAL_PHONENB	dial_phonenb	STRING
WIP_BOPT_DIAL_RINGCOUNT	dial_ringcount	NUMBER
WIP_BOPT_DIAL_SPEED	dial_speed	NUMBER
WIP_BOPT_ERROR	error	NUMBER
WIP_BOPT_GPRS_HEADERCOMP	gprs_headercomp	BOOL
WIP_BOPT_IP_ADDR	ip_addr	ADDR
WIP_BOPT_IP_DNS1	ip_dns1	ADDR
WIP_BOPT_IP_DNS2	ip_dns2	ADDR
WIP_BOPT_IP_DST_ADDR	ip_dst_addr	ADDR
WIP_BOPT_IP_GW	ip_gw	ADDR
WIP_BOPT_IP_NETMASK	ip_netmask	ADDR
WIP_BOPT_IP_SETDNS	ip_setdns	BOOL
WIP_BOPT_IP_SETGW	ip_setgw	BOOL
WIP_BOPT_LOGIN	login	STRING
WIP_BOPT_NAME	name	STRING
WIP_BOPT_PASSWORD	password	STRING
WIP_BOPT_PPP_CHAP	ppp_chap	BOOL
WIP_BOPT_PPP_ECHO	ppp_echo	BOOL
WIP_BOPT_PPP_MSCHAP1	ppp_mschap1	BOOL

WIP_BOPT_PPP_MSCHAP2	ppp_mschap2	BOOL
WIP_BOPT_PPP_PAP	ppp_pap	BOOL
WIP_BOPT_RESTART	restart	BOOL

Fields marked as having type ADDR take a string representing a numerical IPv4 address, such as "192.168.1.4".

All of these fields can be accessed by reading. For instance, if you want to know the address associated with the bearer on UART2, you can type on the shell:

```
$ =proc.bearers.UART2.ip_addr
= "192.168.1.4"
$ _
```

Notice that most of these field must be set when the bearer is opened, i.e. through the options argument of `wip.bearer_client()` or `wip.bearer_server()`.

3.3.2 channels

- Channels share a common Object Oriented API: each kind of channel has its own constructor, but once created, all are handled with the same methods. Besides, options getting and setting are handled as regular fields.

Options

Options are reified as regular fields, which can be read e.g. with `myTcpSocket.port`, and set with e.g. `myTcpSocket.ttl=255`. The list of supported options is given in the table below. The "Lua type" column indicates the kind of data that can be read or written for these options:

- NUMBER: an integer number
- STRING: a string
- BOOL: any value. `false` and `nil` are interpreted as false, everything else as true.
- ADDR: an IP address of the form "`nnn.nnn.nnn.nnn`". You can't provide DNS-resolved address to these fields, but most of the time, to an `xxx_addr` option corresponds an `xxx_straddr` option that accepts DNS-resolved addresses.

WIP option name	Lua field name	Lua type
WIP_COPT_ACCOUNT	account	STRING
WIP_COPT_ADDR	addr	ADDR
WIP_COPT_BOUND	bound	BOOL
WIP_COPT_CHECKSUM	checksum	BOOL
WIP_COPT_DONTFRAG	dontfrag	BOOL
WIP_COPT_ERROR	error	NUMBER
WIP_COPT_FILE_NAME	file_name	STRING
WIP_COPT_INTERVAL	interval	NUMBER
WIP_COPT_KEEPALIVE	keepalive	BOOL
WIP_COPT_NODELAY	nodelay	BOOL
WIP_COPT_NREAD	nread	NUMBER
WIP_COPT_NWRITE	nwrite	NUMBER
WIP_COPT_PASSIVE	passive	BOOL
WIP_COPT_PASSWORD	password	STRING
WIP_COPT_PEEK	peek	BOOL

WIP_COPT_PEER_ADDR	peer_addr	ADDR
WIP_COPT_PEER_PORT	peer_port	NUMBER
WIP_COPT_PEER_STRADDR	peer_straddr	STRING
WIP_COPT_PORT	port	NUMBER
WIP_COPT_RCV_BUFSIZE	rcv_bufsize	NUMBER
WIP_COPT_RCV_LOWAT	rcv_lowat	NUMBER
WIP_COPT_RCV_TIMEOUT	rcv_timeout	NUMBER
WIP_COPT_REPEAT	repeat	NUMBER
WIP_COPT_RESET_CEV_READ	reset_cev_read	BOOL
WIP_COPT_RESET_CEV_WRITE	reset_cev_write	BOOL
WIP_COPT_SND_BUFSIZE	snd_bufsize	NUMBER
WIP_COPT_SND_LOWAT	snd_lowat	NUMBER
WIP_COPT_STRADDR	straddr	STRING
WIP_COPT_TOS	tos	NUMBER
WIP_COPT_TRUNCATE	truncate	BOOL
WIP_COPT_TTL	ttl	NUMBER
WIP_COPT_USER	usre	STRING

Events

channels can emit events “open”, “read”, “write”, “peer_close”, “error”, “ping”, “done”. However, most of time it isn’t required to deal with them: channel creation functions generally wait for “open” before returning, so that the channel is operational at the next line of code; :read() and :write() methods transparently put the current task to sleep and wake it up when required, so that they can be used without having to deal with events.

For instance, you can create a TCP client channel then wait for it to emit “error” or “peer_close” with:

```
myTcpChannel = wip.tcp_client(SOME_SERVER_ADDRESS, 80)
wait(myTcpChannel, {“error”, “peer_close”})
```

Examples

Here is how to perform an HTTP query, directly and interactively, with a TCP socket. You need a connection to internet, e.g. through GPRS.

```
$ x = wip.tcp_client ("www.google.com", 80) -- Establish connection
$ x:write "GET / HTTP/1.0\r\n\r\n"           -- Send hand-made request
$ = x:read()                                -- Check response
= "HTTP/1.0 302 Found\r\n […]</HTML>\r\n"
$ x:close()                                 -- Clean up
```

Similarly, here is how to perform a simple active FTP get: the content of file `data.txt` is retrieved in variable `data`. Everything is hard-coded, and no error handling is done. This is for illustration purpose only: if you want to actually perform FTP transactions, use `wip.ftp_client()`!

```
x = wip.tcp_client ("192.168.1.5", 21) -- x :: control socket, see RFC959
y = nil                                -- y :: data communication socket
-- z :: data server socket. First connection socket is put in y,
-- then z is closed in the acceptation callback.
z = wip.tcp_server (1024, function(client) y=client; z:close() end)
z:wait "accept" -- wait for connection
x:write "USER anonymous\r\n"
x:write "PASS lua@wavecom.com\r\n"
x:write "PORT 192,168,1,4,4,0\r\n"
x:write "RETR data.txt\r\n"
data = y:read "*a" -- read everything on y until closed by peer
x:write "QUIT\r\n"
y:close(); x:close() -- clean up
```

`loadchannel(channel)` ASYNC

Read everything on a channel. If it's Lua bytecode, undump it to return the corresponding function. If it isn't, consider it as source code, compile it, and return the corresponding function.

This call is optimized w.r.t. memory: it destroys data while compiling it, thus allowing compiling bigger chunks than a full reading as a single string, followed by a single-pass compilation.

`CHANNEL:read([x])` ASYNC

Read and return data, as a string, from `CHANNEL`. Its exact behavior depends on **`x`**:

- **`x=="*l"`:** read a line, terminated by regular expression `wip.EOL_PATTERN` (by default, "\n") or by the peer closing of the channel; might block the current thread and wait for more data if required.
- **`x=="*a"`:** read the whole channel, i.e. returns all data at once when the channel is remotely closed. Block the current thread if necessary. In most cases you should prefer "`*ab`" (cf. below).
- **`x=="b"`:** return all currently bufferized data, as a buffer, i.e. an array of strings. Buffers are preferable to strings for long pieces of data (more than

a couple KB), as they don't stress the memory allocator by requesting huge, single-piece chunks of heap memory.

- **x=="*lb":** read a line, blocking the thread if necessary, and returns it as a buffer (array of short strings, cf. above)
- **x=="*ab":** blockingly read all of the channel, as "*a" does, but returns is as a buffer rather than a single string.
- **x==<nothing>:** returns whathever data is currently available, without blocking, as a single string.
- **x == <number>:** if x is a number, read x bytes of data, if necessary blockingly.

CHANNEL:write(…) ASYNC

write all its arguments, blocking the thread if required by a full internal buffer. Arguments can be strings or numbers (which are then transformed into strings). It is also acceptable to give a single buffer (array of strings, as returned by some :read() calls) as an argument.

CHANNEL:get(filename) ASYNC

Wrapping around wip_getFile(): applied on a filesystem channel, returns a file download channel.

CHANNEL:put(filename) ASYNC

Wrapping around wip_putFile(): applied on a filesystem channel, returns a file upload channel.

CHANNEL:shutdown(x, ['read'], ['write'])

Shutdowns a channel in read and/or write direction(s).

CHANNEL:load() ASYNC

Equivalent to loadchannel(CHANNEL)

CHANNEL:wait(events) ASYNC

Equivalent to wait({CHANNEL}, events)

CHANNEL:hook(events, hook [,args])

Equivalent to sighook({CHANNEL}, events, hook, args)

CHANNEL:state()

Return the channel's state.

CHANNEL:close()

Close and release the channel. Think also about niling all variables referring to that channel, so that it can be garbage collected:

```
x:close(); x=nil
```

A Very common idiom for closing a channel is to do this in a signal hook. Channel handling happens in an infinite loop in a separate thread, and doesn't take care of errors nor peer shutdowns. When an error or shutdown is caught by the signal, the channel is closed, and the handling thread (presumably blocked on a :read() or a :write()) is killed by the induced error (a read/write causes an error when the channel is closed).

```
local channel = ...
channel:hook(function() channel:close(); channel=nil end)
run(function()
       while true do
          do_stuff_with (channel)
       end
    end)
```

3.3.3 tcp

wip.tcp_client(host, port [,opts]) ASYNC

Create a TCP client channel connected to host (address given as a string) on port port. opts is a table associating option names to option values; these option pairs will be passed to the internal wip_tcpClientCreateOpts() call.

This function blocks until the connection succeeds or fails (event WIP_CEV_OPEN or WIP_CEV_ERROR), so that the resulting channel is immediately usable.

FIXME: handling of opts by constructor not implemented yet.

wip.tcp_server(port, [accept_function])

Create a TCP server listening on local port port. If a function accept_function is provided, it will be used to create a new thread every time a connection request is accepted by the server. In this thread, accept_function will receive as arguments the newly created communication channel, and the server channels which accepted the connection.

This makes handling of multiple connections by TCP servers fully automatic. An idiom to accept only one client and write the code linearly is:

```
local client
wip.tcp_server(123, function(c,s) s:close(); client=c end):wait "accept"
-- 'server' is closed and 'client' contains the only accepted connection.
-- From here, use 'client' as you wish.
```

Examples

This program diverts debug traces to every sockets which connect to port 2007, by:

- creating a list of subscribed sockets called log_sockets
- every time a new socket is accept it, insert it into log_sockets
- replace trace() by a function which writes to all working sockets in log_sockets.

```
log_sockets = { }
wip.tcp_server(2007, function(c) table.insert(log_sockets, c) end)

function trace(...)
   for _, c in ipairs(log_sockets) do
      if c:state()=="ready" then c:write("\r\n", ...) end
   end
end
```

Notice that a cleaner way to do this would to close sockets on error, and use log_sockets as a set by putting values in it as keys rather than values; this is a rather common idiom in Lua:

```
log_sockets = { }

local function on_accept(c)
   log_sockets[c] = true
   local function cleanup() c:close(); log_sockets[c] = nil end
   c:hook({'error','peer_close'}, cleanup)
end

wip.tcp_server(2007, on_accept)

function trace(...)
   for c, _ in pairs(log_sockets) do c:write("\r\n", ...) end
end
```

wip.ftp_client(host [,opts]) ASYNC

Creates an FTP client channel, on which get() and put() can be called to transfer files. It blocks the current thread until the FTP login is fully performed or failed, i.e. there's no need to wait for event "open".

opts is a name → value hashtable, which can contain:

- `user=<string>`: username for login (default = anonymous)
- `password=<string>`: password (default=wiplua@wavecom.com)
- `port=<int>`: FTP server port (default=21)
- `mode=<bool>` : `true` → passive, `false` → active. Default=active

Once the session is open, files can be read/written with methods `:get()` and `:put()`.

wip.http_client(?)

Not implemented

wip.snmp_client(?)

Not implemented

wip.pop3_client(?)

Not implemented

3.3.4 udp

wip.udp([ip_address, port])

Create a UDP socket, optionally pseudo-connected to the given address and port.

wip.udp_sendto(buffer, [ip_address, port])

Send a datagram containing `buffer` to port `port` at address `ip_address`.

FIXME: not implemented

wip.udp_recvfrom()

Read a datagram, and returns its content, its sender, and the originating port.

FIXME: not implemented

3.3.5 ping

wip.ping()

FIXME: not implemented

3.3.6 fcm

FCM flows are implemented through a channel API: once created, they respond to methods `:read()`, `:write()` and `:close()` as a WIP channel would.

wip.fcm(name) ASYNC

Return an FCM stream as a channel. If the corresponding FCM is of modem-type, it goes to data mode until the channel is closed with method :close(). name must be one of "UART1", "UART2", "UART11", "UART12", "UART13", "UART14", "UART21", "UART22", "UART23", "UART24", "USB", "GSM", "GPRS".

3.4 Flash objects

Flash objects are reified as tables, i.e. one can read and write them as regular tables, and these operations are transparently translated into flash objects read/write operations. However, these reified tables present a couple of limitations:

- They don't support base functions `next()`, `pairs()`, `ipairs()`
- They can't store functions with upvalues (i.e. with local variables which escaped their scope, see Lua ref manual for details. Upvalues are an advanced feature which doesn't exist in most languages, so don't bother too much: "normal" functions get serialized just fine).
- They can't store userdata (bearers, channels, and other C live objects lifted to Lua)
- Values in tables have to be storable
- Keys in tables have to be storable, and shouldn't be tables not functions. If they are, copies of them will be generated, which is probably not what you want.

A flash table can be translated into a regular (a.k.a. fully reified) table with `flash.reify(flashtable)`; however, operations on the resulting table aren't automatically committed to the flash objects anymore.

Moreover, "raw" tables are also supported: their keys can only be integers, and their values can only be strings, but they directly map the regular ADL flash tables, which eases communication between C and Lua code: a realtime data gathering interrupt can simply fill a flash table, which can be read, processed and reported by Lua code at a later time, without worrying about reboots.

`flash.create(name [,size])`

`rawflash.create(name [,size])`

Create a new flash table, referenced by `name`. It has `size` elements (defaults to 256). Return the created table.

`flash.load(name)`

`rawflash.load(name)`

Return the flash table or raw flash table referenced as `name`. Cause an error if the table doesn't exist.

`flash.reify(flashtable)`

Create a copy of a flash table as a regular table. The copy supports enumerators and metatable hacks, but modifications on the copy won't be reflected in flash anymore.

`flash.collect(flashtable)`

Optimize the internal representation of a flash table which experienced many entry deletions.

About flash handling, see also `save()` and `loadboot()` in the misc. subsection.

FIXME: enumerators `ipairs()`, `pairs()` and `next()` should be modified so as to support metamethods. This way, flash tables could behave completely normally.

3.5 at

Access to AT commands and AT unsolicited events instantly gives access to a lot of features: GPIO, phonebook, SMS repository, audio... The main advantage of using AT commands through Lua rather than directly is that you get access to Lua features such as control logic (loops, conditional, functions, vars...), library functions (e.g. regular expressions parsing), blocking calls (no need to mess with callbacks) and multithreading.

The only drawback is speed. If your application requires higher performances, you might create a C function over ADL, bind it to Lua, or use it in a separate, pure C task/interrupt if you're under hard realtime constraints: garbage collected programming is generally a bad idea when it comes to hard real time anyway.

`at(cmd, …)` ASYNC

Run an AT command string, and returns all intermediate responses, as well as the final response, in a table. This call blocks until the final response is received, and queues commands in case of concurrent accesses. In case you want a lower level access for special commands (e.g. `ATD`), see `internal.at_cmd()`.

`cmd` is actually a `printf`-like format string: if it contains codes such as `%i`, `%s` etc., some extra args can be provided.

AT unsolicited

Unsolicited events are reported by emitter "at". The event is the string between the "+" and the ":". Anything between ":" and "\r\n" is passed in an extra signal arg.

Examples

In order to receive keyboard events as unsolicited AT events, we need to call "AT+CMER=,1":

```
at 'at+cmer=,1'
```

Now every keyboard event will be signalled by "+CKEV: m,n", with m the key identifier, and n being 1 for press, 0 for release. We can display these events in a loop:

```
again=true
while again do
  local emitter, event, params = wait("at","CKEV")
  local key, dir = params:match '(%d+),(%d+)'
  key = tonumber(key)
  dir = dir=='0' and 'released' or 'pressed'
  printf("key %i has been %s", key, dir)
end
```

(This loop can be put in the background, and stopped with `again=false`)

Notice that cutting arguments into relevant parts is generally very easy with regular expressions and the `:match()` string method. However, beware that numeric parts are kept as strings, not numbers. Here, key is a string, e.g. "18", unless it's converted to a number with `tonumber()`. This is a frequent cause of bugs.

3.6 sms

sms.send(phonenb, message) ASYNC

Send an SMS in text mode; the SIM card must be ready.

SMS reception event

SMS are signalled by the emitter "sms" sending the event "read". As an attached argument is a table with fields phonenum, timestamp and text, all of them containing a string.

Example

```
function on_sms_rcv (emitter, event, msg)
   assert (emitter=='sms' and event=='read')
   printf( "SMS!\nFrom %s, received at %s:\n%s",
           msg.phonenum, msg.timestamp, msg.text)
   return 'again' -- don't detach the hook.
end
wip.sighook ("sms", "read", on_sms_rcv)
```

Stored SMS handling

Currently through AT commands, a proper consistent API will be provided.

Example

```
$ = at'at+cmgl="REC READ"'
= {
  "",
  "+CMGL: 2,\"REC READ\",\"123\",,\"07/07/25,16:13:32+00\"",
  "Repondeur: 3 nouveaux messages; dernier appel 16:13 du 0687212437.
Rappelez 123.",
  "",
  "+CMGL: 12,\"REC READ\",\"3000\",,\"99/12/31,23:59:59+00\"",
  "",
  "",
  "\r\nOK" }
$ _
```

3.7 shell

The shell allows you to take control of the module, over TCP/IP (telnet) or over some FCM stream such as UARTs. It supports optional authentication, as follows.

If there is no table `wip.shell.auth`, the shell immediately accepts all conncetions. If there is such a field, it's expected to have a field `method`, set at "`open`" (no authentication), "`basic`" (password, sent in clear over the network), or "`challenge`" (challenge/response based authentication: it requires a response generator in addition to knowledge of the password, but makes sure the password can't be eavesdropped).

Login and password information are stored as follows: if there's a `wip.shell.auth.users` table, it's expected to contain user names as keys, and as values, either the password as strings, or a table with a `password` field holding the password. If there is no `wip.shell.auth.users` table, then there should be a `wip.shell.auth.password`, and the authentication dialog will ask for the password without a login.

Example

The following code will make authentication mandatory, with user 'wipuser' and password '123456':

```
wip.shell.auth = {
   method = 'basic',
   users = { wipuser = "123456" } }
```

FIXME: challenge/response authentication not implemented yet.

`wip.shell.channel : channel or nil`

The WIP channel over which the shell is served. Should be nil when state is "`dead`".

`wip.shell.snatch: Boolean`

Whether it is allowed for a TCP client to snatch the shell from another client.

FIXME currently SNATCH instead of snatch

FIXME currently mostly broken

`wip.shell.prompt: string`

The string served as the normal prompt. Default="$".

FIXME should be allowed to make it a function.

wip.shell.prompt2: string

The string served as a go-on prompt, when the command is multi-lines and not terminated. Default=”+”.

FIXME should be allowed to make it a function.

wip.shell.greeting: string

String served as a welcome MOTD when loggin in the shell.

wip.shell.telnet_server([port])

Starts a telnet server to allow connecting to the shell.

FIXME: port is currently mandatory

wip.shell.fcm(name) ASYNC

Directs the shell on an FCM device. Name is an FCM name. For instance, if you are entering AT commands on UART1, you can turn the terminal into a lua shell with:

```
AT+LUA=”wip.shell.fcm ‘UART1’”
Lua Interactive Shell
$ _
```

wip.shell.print(…) ASYNC FIXME Keep it private?

wip.shell.quit() FIXME Keep it private?

3.8 proc

proc.tasks.running: thread or nil

proc.tasks.waiting: (emitter → event → thread_reference) table

proc.tasks.ready: thread list

Threads not running, but ready to be scheduled (not waiting for any event)

proc.channels: (id number → channel) table

All currently running channels, indexed by a unique number

proc.bearers: (id number, name → bearer) table

All currently running bearers. Each bearer is referenced twice: under a unique number id, and under its user-friendly name.

proc.timers: FIXME

Pending and cyclic timers.

3.9 misc

try(f [,args])

Runs `f(args)` in a separate task, and waits for it to terminate. Return `true` followed by all the results returned by `f()` upon success, or `false` followed by the error message upon failure.

This function replaces `pcall()` from the standard library in most use cases, as it allows `f()` to yield coroutine, i.e. call ASYNC functions. It is slower and uses more memory than `pcall()`.

gc()

Perform a full garbage collector, collect the garbage in the `proc.tasks` table, and return the number of bytes taken by Lua VM after collection.

l(…)ASYNC

Load files from the FTP server whose name is in global variable PEER_ADDR. There can be many names, which will all be loaded in a single connection. If the file name doesn't terminate in ".lua" or ".luac", a ".lua" extension is automatically added.

Files must be lua sources or lua bytecode dumps. They are compiled (or undumped) and executed.

When no names are provided, the last series of names is reused. This series is stored in flash, and is therefore remembered across reboots.

save(…)

Take variable names (as strings), and save every global variables named into the flash table wiplua.boot, so that they will be available across reboots.

Example

```
$ y = { 'foo' }
$ x = {1, 2, 3, y, y, f = function(x) return x+1 end}
$ save 'x'
```

<reboot>

```
$ = x
= { 1, 2, 3, { 'foo' }, { 'foo' }, f = [function 0x180c20bc] }
$ = x[4] == x[5] -- Check that shared subexpressions are respected
= true
$ = x.f(3)
= 4
$ _
```

p(…) ASYNC

Print Lua values on the shell: not only strings and numbers, but also arbitrarily nested tables, with indentation. Handle recursive tables gracefully.

4 C API

4.1 Lua state handling

luaW_start(luaL_Reg *init_table);

Start running the Lua VM, running all initialization functions in init_table (see Lua manual for luaL_Reg* init tables).

luaW_run(char *src);

Run src as a new task in the VM. src can be Lua sources ar a compiled chunk.

luaW_stop();

Kill the Lua VM and release associated resources.

luaW_atLuaCmdSubscribe();

Register the AT command AT+LUA="xxx", which will run the Lua source code between quotes in a new task. The VM must be running.

Module initialization functions

int luaopen_strict (lua_State *L);

Strict global variables declaration: with this module loaded, you can't implicitly create a new global variable inside a function. You need to create it at the toplevel, outside any function, with e.g. myGlobalVar = someValue, or to declare it as global with global 'myGlobalVar'.

This catches a lot of bugs linked to typos, and should be kept on unless you're extremely tight on memory.

int luaopen_scheduling (lua_State *L);

Mandatory: loads scheduling primitives.

int luaopen_options (lua_State *L);

Mandatory for all WIP libraries: handles channel, bearer and stack options.

int luaopen_channels (lua_State *L);

WIP generic channels. Requires luaopen_options. Required by luaopen_tcp, luaopen_shell.

int luaopen_tcp (lua_State *L);

WIP networking channels: TCP, UDP, PING, FTP, HTTP, SMTP, POP3. Requires luaopen_channels and luaopen_options.

```
int luaopen_flash_read  ( lua_State *L);
```

Flash tables reading.

```
int luaopen_flash_write ( lua_State *L);
```

Flash table creation and writing. Requires luaopen_flash_read.

```
int luaopen_mem         ( lua_State *L);
```

Memory monitoring functions.

```
int luaopen_sms         ( lua_State *L);
```

SMS monitoring and sending.

```
int luaopen_timer       ( lua_State *L);
```

Timer events.

```
int luaopen_shell       ( lua_State *L);
```

shell over channels, be it TCP or FCM. Requires luaopen_channels. You probably want luaopen_fcm and/or luaopen_tcp as well.

```
int luaopen_bearers     ( lua_State *L);
```

WIP bearers. You want this if you plan to handle bearers from Lua.

```
int luaopen_print       ( lua_State *L);
```

Advanced printing. Without this, when trying to print a table, you'll simply get something like "[table 0x12345678]".

```
int luaopen_misc        ( lua_State *L);
```

Mandatory, it includes boot code.

```
int luaopen_at          ( lua_State *L);
```

AT commands handling.

```
int luaopen_rawflash    ( lua_State *L);
```

Raw flash tables, interoperable with adl_flhXxx() functions.

4.2 WIP interface from C

channels and bearers reification / signal emission / idioms for handling multitasking.

Channel handling

```
wip_channel_t luaW_checkchannel( lua_State *L, int i);
```

Check that a function's argument number i is a channel and returns it, or causes an invalid parameter error.

```
void luaW_newchannel(   lua_State *L, wip_channel_t c);
```

Register a channel into Lua. A channel must be registered exactly once.

```
int luaW_pushchannel(  lua_State *L, wip_channel_t c);
```

Push an already registered channel on Lua stack.

Bearer handling

```
wip_bearer_t luaW_checkbearer( lua_State *L, int i);
```

Check that a function's argument number i is a bearer and returns it, or causes an invalid parameter error.

```
void luaW_newbearer(   lua_State *L, wip_bearer_t b, char *name);
```

Register a bearer into Lua. A bearer must be registered exactly once.

```
int luaW_pushbearer(  lua_State *L, wip_bearer_t b);
```

Push an already registered bearer on Lua stack.

Signals and synchronization

```
void luaW_signal_channel( lua_State *L, wip_channel_t emitter, char *event,
                          int nargs);
```

Signal event on behalf of channel `emitter`; this signal will be caught By all Lua threads and hooks waiting for it. Some extra arguments can be sent by setting `nargs>0` and stacking the corresponding values on `L`'s pseudo-stack.

`void luaW_signal_str( lua_State *L, char *emitter, char *event, int nargs);`

Signal event on behalf of string `emitter`; this signal will be caught By all Lua threads and hooks waiting for it. Some extra arguments can be sent by setting `nargs>0` and stacking the corresponding values on `L`'s pseudo-stack.

`void luaW_evh_signal( wip_event_t *ev, void *ctx);`

This WIP event handler will emit signals `open / read / write / peer_close / ping / error` on behalf of `ev->channel`. This is the standard way to interface a channel with Lua.

4.3 Optimized memory pools

Although not mandatory, reserving and configuring a memory area for Lua will significantly improve memory efficiency. However, even when memory pools are set up, it falls back on ADL memory allocator when it is exhausted, or can't handle a certain size of memory chunk.

FIXME

5 Samples showcase

5.1 HTTP server

Among the samples provided

5.2 FTP server

5.3 XML encoder and decoder

5.4 Multi-protocol application

Send commands through SMS, through V24, through TCP/IP, through HTTP...

Send UART data through FTP (without using FTP nor WIPSoft)

```
at+lua="wip.bearer_client('GPRS',{…})"
at+lua="uart2ftp 'wipuser:123456@ftp.wavecom.com/mydata.txt'"
<DATA><ETX>
OK
```

FIXME: untested code.

```
function uart2ftp(url)

  -- Parse URL
  local user, pass, server, file = url:match '^(.-):(.-)@(.-)/(.*)$'
  if not user then
    user, pass = 'anonymous', 'lua@wavecom.com'
    server, file = url:match '^(.-)/(.*)$'
  end
  if not server then error "Can't parse URL" end

  -- Connect to FTP server
  local session = wip.tcp_client (server, 21)
  local data = nil
  local s = wip.tcp_server (0, function(client) y=client; s:close() end)
  z:wait "accept"
  local a,b,c,d = x.addr:match '(%i+).(%i+).(%i+).(%i+)'
  local e,f     = x.port/256 + x.port%256
  data:write(string.format(
```

```
    "USER %s\r\nPASS%s\r\nPORT %s,%s,%s,%s,%i,%i\r\nRETR %s\r\n",
    user, pass, a, b, c, d, e, f, file))
  repeat local line = session:read '*l' until line:match '^227 '

  -- Pipe data
  local uart = wip.fcm 'UART1'
  repeat
    local finished = false
    local x = uart:read()
    local i, j = x:find '.-[^\10]\3'
    if j then x = x:sub(1, j-2); finished=true end
    x=x:gsub('\10(.)','%1')
    uart:write(x)
  until finished

  -- Clean up and leave
  uart:close()
  data:close()
  session:write 'QUIT\r\n'
  session:close()
end
```

6 Other tools

6.1 off-line compilation

Compiling an application on the WCPU is very convenient, but comes with a couple of drawbacks:

- you need to download it before knowing whether there are any errors in it, including trivial syntax errors
- Compilation takes time
- More important, it also takes RAM. On a WCPU with only 256 KB of RAM, one must sometimes be careful with memory usage.
- The final compiled program contains debug information, which can double its memory footprint.

For these reasons, we offer a compiler that turns source code into bytecode on the development computer. It also has a -s option which strips debug information from the resulting binary. This compiler is actually nothing but a regular Lua compiler for Intel processors with the number type set to 32 bits integers (ARM and Intel processors already have the same endianness). This compiler is called `luac51int.exe`, and consists of this single executable file.

The Lua source to bytecode compiler provided with the distribution is called `luac51int.exe`, and behaves as the regular luac.exe compiler except that it compile for the integral version of Lua, embedded in OpenAT-Lua.

lua51int.exe is an MS-Windows interpreter for integral Lua: it can read Lua bytecode intended for OpenAT-Lua; however, most OpenAT-Lua programs won't run, since OpenAT libraries are not available under MS-Windows.

lua2c.bat uses the binaries above to compile lua source files into C source files holding a const char * declaration, storing precompiled bytecode. The resulting file contains that string, and an integer constant holding its length. These can be used with luaW_lrun(L, string, string_length) to be run byt the Lua VM.

Option -h gives a short usage manual:

```
C:\workdir> lua2c -h

Compile a lua source file or a luac bytecode file into a C source file
declaring a global const string, and a const int holding that string
length. The string can be run by a lua interpreter, e.g. luaW_lrun().

Usage: lua2c input_file [options]

Options:
   -o <filename>: output C file name.
   -n <name>:     name of the string global variable
   -l <name>:     name of the string length global variable
   -L <number>:   how many chars to print per line in the C file
   -C <filename>: lua compiler used to produce bytecode, if applicable
   -c / +c:       whether it is a lua source file to precompile
   -s / +s:       whether to strip debug info when precompiling
   -a / +a:       whether result shall be appenned to C file or overwrite it
   -v / +v:       whether to print out configuration
   -h:            print this help and exits

For options which are not set explicitly, sensible defaults are
guessed. Check them with -V in case of doubt.

C:\workdir> type hello.lua

print "hello world"

C:\workdir> lua2c -s hello.lua
C:\workdir> type hello.c

/* This file contains a precompiled Lua chunk, and should not be
 * edited manuallly. Edit the corresponding lua source file and
 * regenerate it instead.
 *
 * This file has been generated with:
 *
 * $ lua51int lua2c.lua -s hello.lua
 */

#pragma warning(disable:4305) /* Obnoxious VC6 warning off. */

static const char  _hello[] = {
   27, 'L', 'u', 'a', 'Q',   0,   1,   4,   4,   4,   4,   1,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   2,   2,   4,   0,   0,   0,
    5,   0,   0,   0, 'A', '@',   0,   0,  28, '@',   0,   1,  30,   0, 128,   0,
    2,   0,   0,   0,   4,   6,   0,   0,   0, 'p', 'r', 'i', 'n', 't',   0,   4,
   12,   0,   0,   0, 'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd',   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0 };
const char *hello = _hello;
const int hello_len = 96;
C:\workdir> _
```

Appendix: Setting up Windows for TCP/IP over PPP Client

This appendix will walk you through the installation of a PPP client over a serial port in Windows XP. Some slight adaptations might be required for other Microsoft OSes.

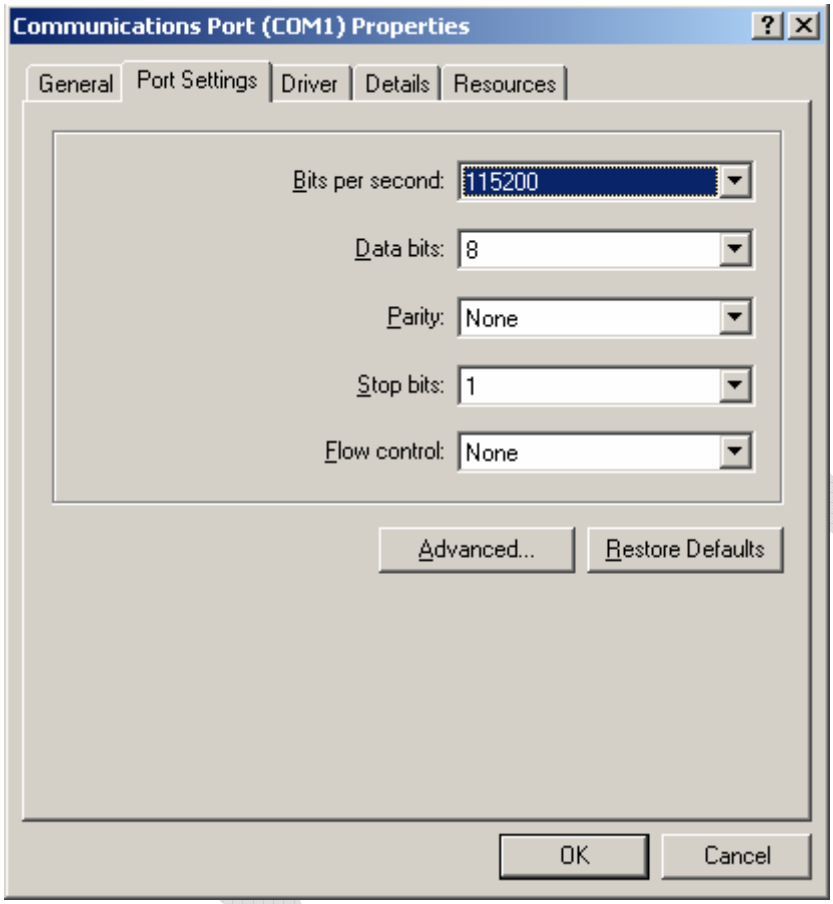
We assume that you have at least one, ideally two serial ports on your development computer. We also assume that this port is connected to a devkit port serving TCP/IP as a PPP server over UART; with the default sample application, that would be UART1. We also assume that this port is able to serve data at 115 Kbps, which is the case of both UARTs on the Q26 and WMP100 devkits.

Important warning: Beware that UART2 on WCPUs miss signals DTR, DSR, RI and DCD. This can confuse some lower end serial ports, especially some serial-to-USB converters. If you experience troubles serving PPP over UART2, try on UART1.

6.2 Configure the serial port

We assume you have connected UART1 to your computer's COM1.

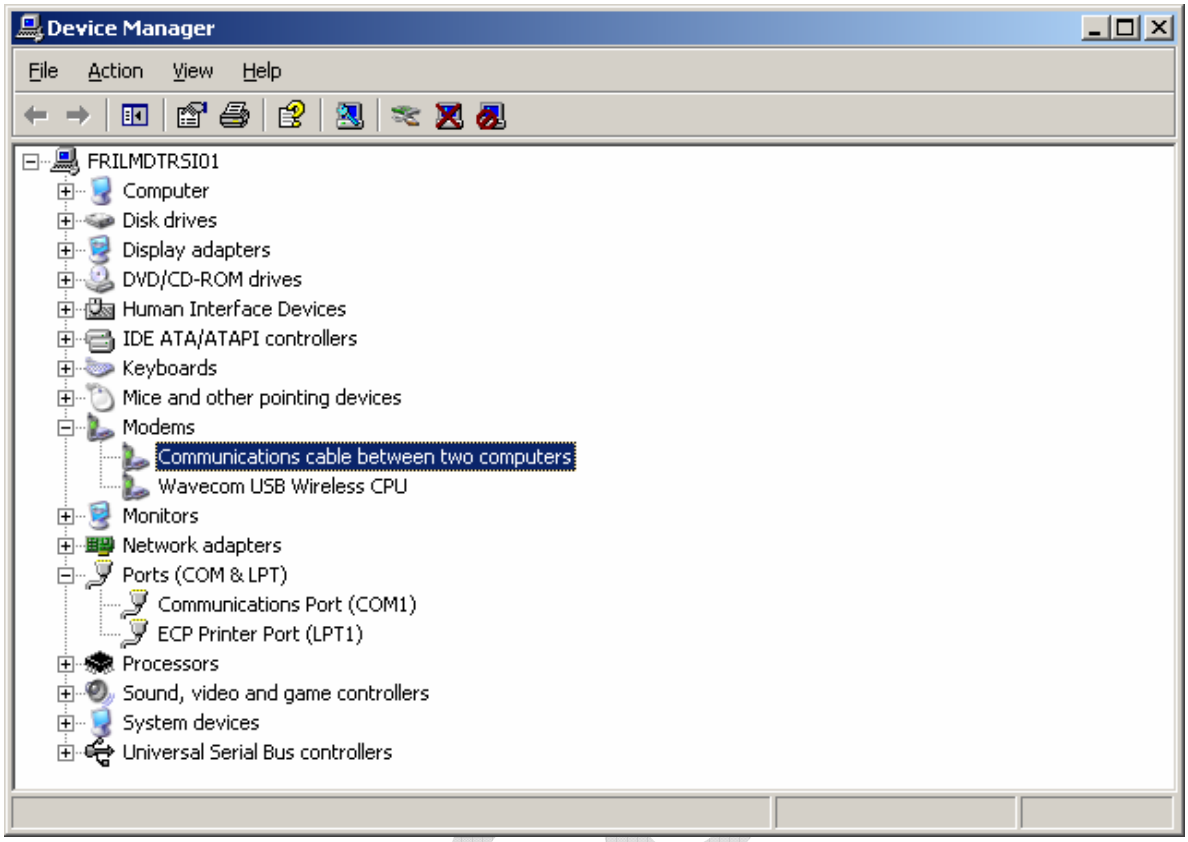
Start > Parameters > Control Panel > System > Hardware > Device Manager > Ports (COM & LPT) > Communication Port (COM1) > right-click > Properties > Port Settings



Set the bit-rate to 115200.

6.3 Add the devkit as a modem

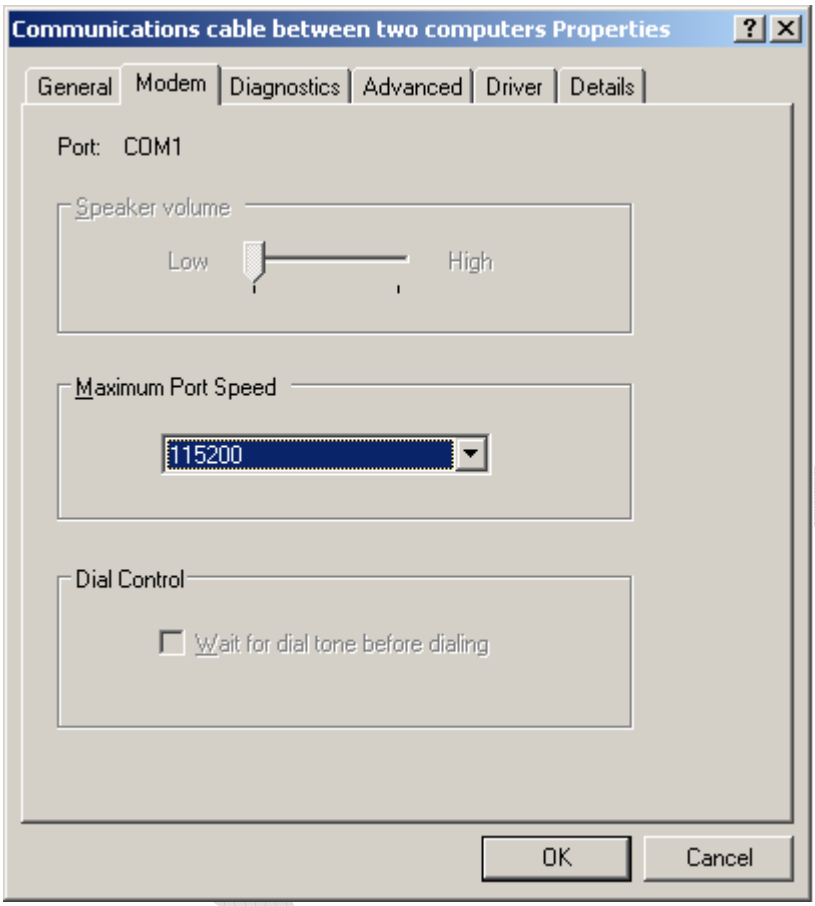
Start > Parameters > Control Panel > Add Hardware > Next > "Yes, I have already connected the hardwatre" > "Add a New Hardware Device" > "Install the Hardware that I Manually select from a list (bottom of the scroll-list) > Modems > "Don't Detect my Modem" > Communication cable between two computers > COM1 > Finish



Check the installation:

Start > Parameters > Control Panel > System > Hardware > Device Manager > Modems > Communication cable between two computers > Right-click > Properties > Modem (2nd tab)

Set maximal speed to 115200:



6.4 Create the connection

Start > Settings > Network Connections > Right-click > Open > Create a new Connection (in the panel on the left) > Setup an advanced Connection > Connect directly to another Computer > Guest "This computer is used to access information on the host computer > [give a name to the connection] > Communication cable between two Computers (COM1) > "Anyone 's use" or "My use only", at your preference > Finish

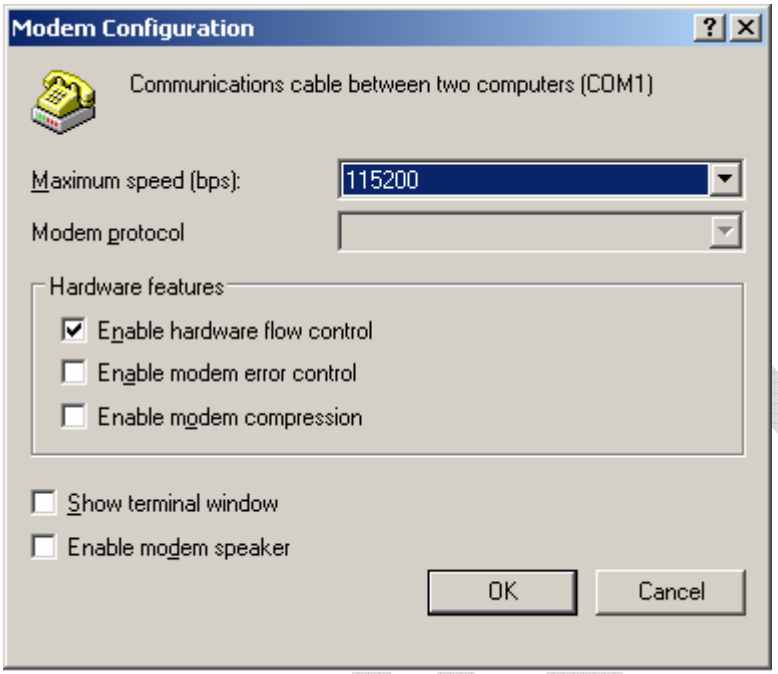
Set username: **wipuser**

Set password: **123456**

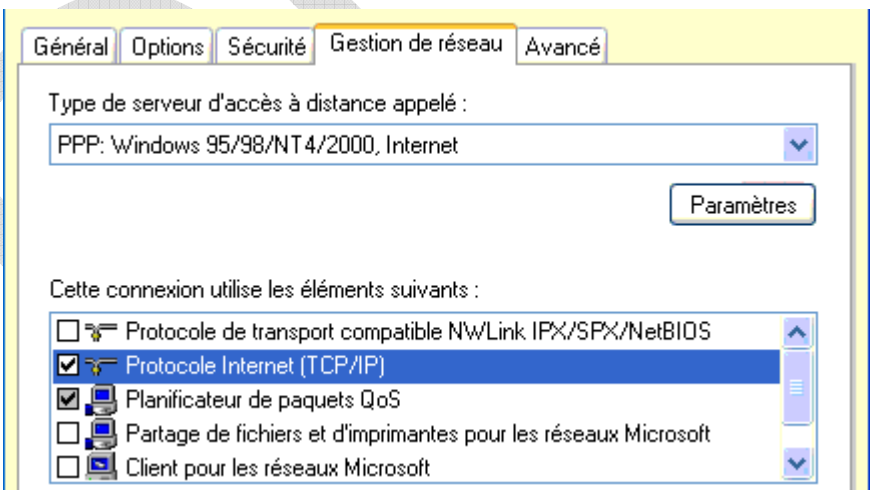
Check the "*remember*" checkbox.

Set the connection's properties

Properties > Configure... (button on the right, below the scroll list) > Set speed to 115200

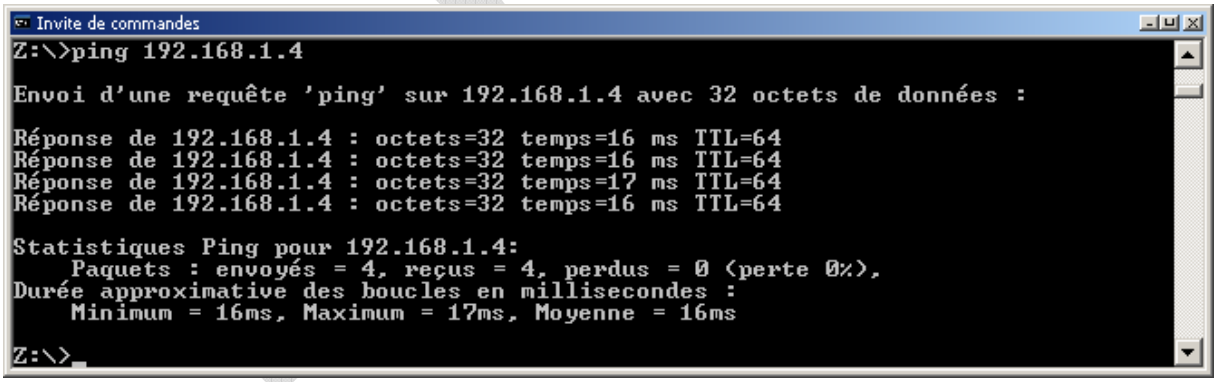


Network config > Check "Internet Protocol (TCP/IP)"



Connect:

That's it; you're connected to your WMP100 through a TCP/IP link. You can check by pingin the module:



7 Appendix: anatomy of a Lua application

Lua is provided as a library, so that you can add it to your programs, and add C functions in your Lua programs.