Lua for RePhone
(Xadow GSM+BLE)

Programming Manual

Ver.: LuaRephone 1.0.9
LoBo 09/2016

table of contents

Installing and using RePhone Lua

Flashing firmware

Before using **Xadow GSM+BLE** as Lua development board, the right firmware has to be flasshed to it.

If your Rephone is already flashed with SEEED02A_DEMO_BOOTLOADER_V005_MT2502_MAUI_11CW1418SP5_W15_29.bin firmware, you can skip this step.

Flash the RePhone firmware using *FirmwareUpdater.exe* included in FirmwareUpdate directory. If RePhone firmware is not already selected, click on Other and select `SEEED02A_DEMO_BB.cfg` in `FirmwareUpdate/firmware/LinkIt_Device/RePhone/W15.19.p2-uart` directory

The purpose of flashing the firmware is to disable the MT2502 debug output and prepare the Rephone USB ports to be used for accessing the application and AT Commands interfaces.

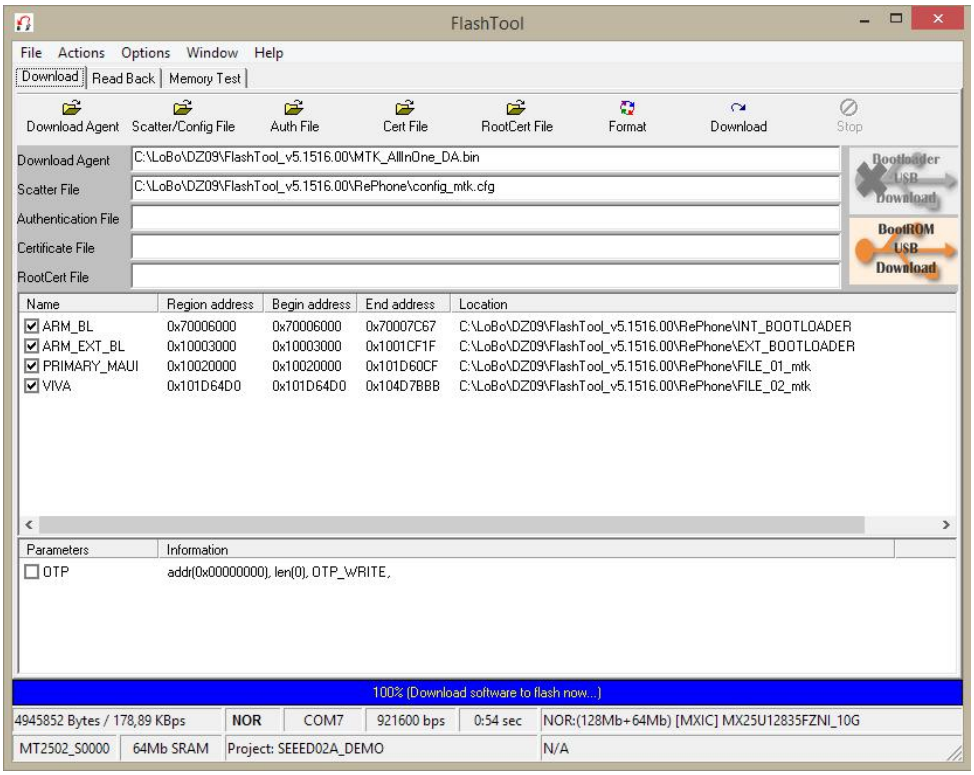
Flashing the RePhone firmware using *FirmwareUpdater.exe* can be tricky as it often cannot successfully detect serial port.

Much easier way to program it is using MTK FlashTool. Version 5.1516.00 must be used, you can download it from
https://github.com/loboris/RePhone_on_Linux/raw/master/FirmwareUpdate/FlashTool_v5.1516.00.zip

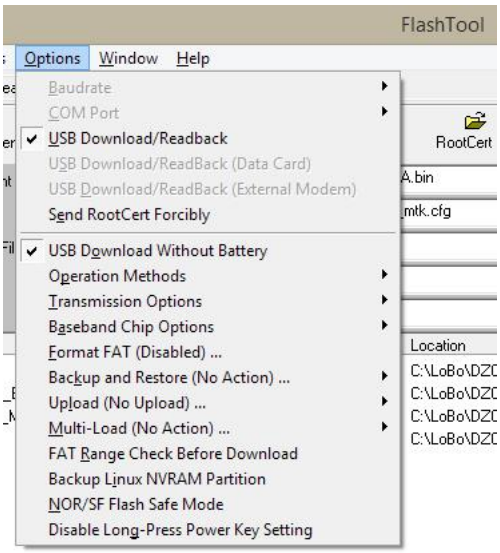
Unpack it to some empty directory on your Windows (virtual) machine. RePhone firmware which can be used with Lua for Rephone is included in “RePhone” subdirectory.

Disconnect the battery from RePhone and unplug the USB cable.

Start ***Flash_tool.exe:***



- Click on "**Download Agent**" button and select "**MTK_AllInOne_DA.bin**" from FlashTool base directory.
- Click on "**Scatter/Config File**" button and select "config_mtk.cfg" from RePhone subdirectory (You can also select any other config file if you are flashing different firmware).
- Click on options and select "**USB Download/Readback**" and "**USB Download Without Battery**" and deselect all other options. You can click on "**Format FAT**" and select if you want to format FAT FS after firmware download or not.



- **Connect the USB cable** (**do not connect the battery**)
- Click on "Download" button
- Rephone will be detected and the download process will begin. After successful download, success window will be shown:

Using on Linux

- To enable RePhone (Xadow GSM+BLE) **mass storage** mode you have to modify `/lib/udev/rules.d/40-usb_modeswitch.rules`. Edit the file (as root), find and comment the line:
  `ATTR{idVendor}=="0e8d", ATTR{idProduct}=="0002", RUN+="usb_modeswitch '%b/%k'"`
  so that it looks like that:
  `#ATTR{idVendor}=="0e8d", ATTR{idProduct}=="0002", RUN+="usb_modeswitch '%b/%k'"`

  If the ***40-usb_modeswitch.rules*** is changed (after some update or distro upgrade) you have to make the change again.

- To prevent Ubuntu accessing RePhone modem interface create the file (as root):
  `/etc/udev/rules.d/98-rephone.rules`
  and place in it the line:
  `ATTRS{idVendor}=="0e8d", ATTRS{idProduct}=="0023", MODE="0666", ENV{ID_MM_DEVICE_IGNORE}="1"`

After making the changes, you can **reload the udev rules** with the command:
`sudo udevadm control --reload-rules`

REBOOT

When Rephone is connected in **mass storage mode**, it's internal drive will appear as /dev/sdX (5MB drive), where X will be different letter on your system.
The drive may appear in your File manager as MEDIATEK FLASH DISK.
You can mount it and transfer the files.

When RePhone is connected in **app mode**, two serial devices will appear:
`/dev/ttyACM0` the application USB serial port interface
`/dev/ttyACM1` MT2502 modem interface (AT Commands)

The **Mobile broadband connection** may appear in **Network Manager**, if it does, **DISABLE IT!**, otherwise Ubuntu will try to communicate to MT2502 modem interface.
Once disabled it will stay disabled, no need to do it every time.

Installing lua.vxp

After the Xadow GSM+BLE is flashed with the right firmware, you have to transfer **lua.vxp** to it. Always use the latest *lua.vxp* from GitHub vxp directory.
To copy the file, connect Xadow GSM+BLE in mass storage mode (connect the battery and then connect USB cable).
Create MRE directory on 5MB drive if it does not already exist. Copy downloaded *lua.vxp* to MRE directory.
In the root directory of the 5MB drive create the file autostart.txt and place in it the following content:

```
[autostart]
App=C:\MRE\lua.vxp
```

Using interactive Lua shell

Using Interactive Lua Shell the user can interactively execute Lua functions and scripts.
The Shell can be accessed via serial interface using terminal program. As default, serial interface is started on USB serial port, but it can be changed later to HW UART or Bluetooth using *sys.retarget()* function.

It is recommended to use terminal emulation program which supports *VT100/ANSI* terminal emulation and file transfer using *ymodem* protocol. On Linux, usinig **minicom** is recommended, on Windows using **Teraterm** or **ExtraPutty** is recommended (standard Putty can be used, but it does not support *ymodem* file transfer).

Setup the terminal emulator as follows:
Serial Port parameters: 115200 bd, 8 bit, no parity, no flow control (when using USB serial, any higher baud rate can also be selected).
Backspace key: Ctrl-H
Use LF as new line character (Putty: “Implicit CR in every LF”; Teraterm: New-line, Receive&transmit = LF;)

When entering text, simple line editing featurers can be used. Left/Right arrow, Home and End keys can be used to navigate through line, Backspace and Del to delete, Enter to accept. Previously entered lines can be accessed using Up/Down arrow keys.
When valid Lua function or chunk is entered it will be executed immediately. Multi line chunks can be entered, all lines after the first will show **>>** prompt.

Simple **file editor** is provided using *term.edit(file_name)* function.
Files can be transferred to/from Lua file system using ymodem file transfer functions.
Special key **Ctrl-D**, if entered on the beginning of the line **reboots** the system.

System parameters

System parameters is special **Lua table __SYSPAR** which can contain the parameters used on system boot or any user parameter.
The parameter must be entered in format: **key=value**.
key is the parameter name, **value** it's value. Value can be number, string or Lua table.

Some special keys are recognized on and some action is taken if found:

apn="gprs_apn"	GPRS apn is set to given value
ntp=tz	*net.ntptime(tz)* function with *tz* timezone is executed to set system date/time from NTP server, apn must also be set
lcdinit=type	*lcd.init(type)* function is executed
termcol=n	*term.setcols(n)* function is executed
termlin=n	*term.setlines(n)* function is executed

Other parameters can be used by user programs.

The table can be **saved** to the special user flash area (not visible externally) and is read on boot.

System variables are special Lua variables which are set on system boot. They are saved to user flash area (not visible externally) with system parameters and are read and set on boot.
For now only **C heap size** and **watchdog timeout** value are used.

Memory

Xadow GSM+BLE has ~1.3 MB of RAM available to user.
Part of that memory is used for *lua.vxp*, system static variables, internal GSM/GPRS/BT functions, internal Lua modules and Lua environment itself.
After the complete system loading some 650 ~ 800 KB of RAM (depending on configuration and number of used lua modules) is available for user programs and data.
Part of user RAM is used for C functions from Lua modules (min 32K, configurable), the rest is used for LUA stack.
This is far more than the common IoT modules have, and more than enough for complex Lua programs.

Boot details

After power up and firmware execution, **lua.vxp** is loaded to RAM end executed.

Memory, system functions and Lua environment initialization as well as hardware setup is done.
System parameters are loaded from special flash area (if found) and predefined functions are executed.

Special lua script files are searched on file system and executed if found:
init.lua or init.lc (compiled init.lua)
autorun.lua or autorun.lc (compiled autorun.lua)

After the execution is finished, interactive Lua shell is started.
Some info about available memory and Lua version is printed on terminal:

```
Lua memory: 577792 bytes, C heap: 65536 bytes
Executing 'init.lua'
LUA SHELL STARTED [ver.: LuaRephone 1.0.6] [2016/08/26]
> Lua memory: 577792 bytes, C heap: 65536 bytes
>
```

After that Lua functions can be executed interactively:

```
> function test(n)
>> local m
>> m = n * n + 2
>> print(n,m)
>> end
> test(7)
7       51
>
> os.list()
C:\
                        MRE     <DIR>
                 result.txt        39 01/01/04 03:33:38
                ftplist.txt     40000 06/24/16 18:16:54
              autostart.txt        31 07/14/16 10:42:48
                 music2.mp3    238237 04/14/16 07:49:36
                     zz.txt     40000 06/25/16 13:21:08
                  xinit.lua      4378 07/06/16 15:14:10
                      @font     <DIR>
            lcddemo_cor.lua      9184 08/15/16 19:33:50
                   test.amr     20250 01/01/04 02:28:46
                   test.waw     23218 01/01/04 02:50:40
                   test.wav    129852 01/01/04 03:04:10
                  music.mp3    102946 09/18/15 14:07:56
                   test.mp3     23036 01/01/04 04:55:26
                 bme280.lua     10746 08/25/16 10:21:24
                   fget.lua       785 08/26/16 12:27:26
                  test1.lua      7991 08/25/16 21:57:16
                    zzz.txt       283 08/25/16 22:09:50
                   zzzu.txt         0 08/25/16 22:11:20
                   init.lua        88 08/26/16 10:56:20
                   test.txt      2084 01/01/04 01:20:28
                  test1.txt      2088 01/01/04 01:28:52
 System Volume Information     <DIR>

Total 655236 byte(s) in 20 file(s)
Drive free: 3565568,  App data free: 957
> function test(n)
```

OS module

All standard Lua os module functions are supported and some additional functions are added:

copy(from_file, to_file)

res = os.copy(from_file, to_file)

Copy file “from_file” to “to_file”. If the destination file exists, it will be overwritten.

Params:
- from_file: string, file name
- to_file: string, name of the new file

Returns:
- res: 0 on success, error code otherwise

mkdir(name)

res = os.mkdir(name)

Create new directory.

Params:
- name: string, new directory name

Returns:
- res: 0 on success, error code otherwise

rmdir(name)

res = os.rmdir(name)

Remove existing directory.

Params:
- name: string, directory name

Returns:
- res: 0 on success, error code otherwise

exists(name)

res = os.exists(name)

Check if the file or directory exists.

Params:
- name: string, file or directory name

Returns:
- res: **1** if file exists, **2** if directory exists, negative error code otherwise

list(filespec)

os.list(filespec)

List content of the file system directory to stdio

Params:
filespec: optional; string, file specification, can contain dir names and wildchars ("MRE*.vxp")
Returns:
None

compile(name)

os.compile(name)

Compile lua source file to bytecode file. Creates ".lc" file with the same base name as lua source file

Params:
name: string, lua source file name, must have ".lua" extension
Returns:
none

shell_linetype(type)

Ltyp = os.shell_linetype(type)

Set or get Lua interactive shell input line type. By default, **advanced** input type is selected where some line editing functions are enabled. Left/Right arrow, Home and End keys can be used to navigate through line, Backspace and Del to delete. Previously entered lines can be accessed using Up/Down arrow keys.

The **simple** input type can be used if using non ANSI/VT100 capable terminal emulator. In this mode the user can only input characters, use Backspace key to delete last char and accept the line with Enter.

To use the advanced type, used terminal emulator must support ANSI/VT100 emulation.

Params:
type: 0: simple type; 1: advanced type; if none is given, returns current type

Returns:
ltyp: current input line type

table2str([list,] tbl)

tstr = os.table2str(tbl)

os.table2str(1, tbl)

Returns string representation of Lua table or lists the table content on terminal.

Params:
list: optional; if 1, list table content on terminal
tbl: Lua table
Returns:
tstr: string representation of the table (if *list*=0 or not given)

sys module

Functions specific to RePhone/Xadow GSM+BLE.

sys.ver()

lv, fh, bd = sys.ver()

Returns version information.

Params:
    none
Returns:

lv	string, lua version
fh	string, firmware host version
bd	string, firmware build date

mem()

lua_used, lua_total, c_heap = sys.mem()

Returns memory information.

Params:
    none
Returns:

lua_used	currently used memory for Lua stack in bytes
lua_total	total memory available for Lua stack in bytes
c_heap	total heap size available for C functions in bytes

battery()

bat = sys.battery()

Returns battery level in %. *ADC module can be used to get more precise battery voltage.*

Params:
    none
Returns:

bat	battery level in % of full charge

ledblink([led_id])

led = sys.ledblink([led_id])

Set or get current system LED blink. System LED blinks once per second. Any of the RGB leds can be selected.

Params:
led_id optional; LED gpio pin,
predefined constants REDLED, BLUELED, GREENLED can be used
Value 0 can be used to disable LED blink
Without parameter returns current led used.
Returns:
led currently used LED

usb()

res = sys.usb()

Returns the USB cable status, connected or not.

Params:
none
Returns:
res USB cable status: 0 not connected; 1 connected

wdg([wdg_tmo])

res = sys.wdg([wdg_tmo])

Set or get watchdog timeout.
Watchdog timer can be set to the values 10 ~ 3600 seconds. The value is saved in system parameters and *sys.saveparams()* function must be executed for new value to be remembered. The new value will take effect after next reboot.
If called without parameters, the current wdg timeout is returned.

Params:
wdg_tmo optional; watchdog timeout in seconds
Returns:
res current or new watchdog timeout in seconds

c_heapsize([size])

res = sys.c_heapsize([size])

Set or get C heap size.
The heap size used for C functions can be set to the values 32K ~ 256K in 32K increments. The value is saved in system parameters and *sys.saveparams()* function must be executed for new value to be remembered. The new value will take effect after next reboot.
If called without parameters, the current C heap size is returned.

Params:
Size: optional; C heap zize in bytes; will be rounded to 32K
Returns:
res current or new C heap size in bytes

noacttime([noact_tmo])

res = sys.noacttime([noact_tmo])

Set, reset or get no activity timeout.
If no activity is detected in Lua shell (no user input, no program activity), the system is

shutdown after no activity timer expires.
If called without parameters, the current no activity timeout is returned.
Params:
noact_tmo optional; > 0 set no activity timeout in seconds
0 reset no activity timeout
no parameter: return current value
Returns:
res current or new no activity timeout in seconds

shutdown()

sys.shutdown()

Shutdown system.
If wakeup interval is defined, system wakeup will be automatically scheduled to next interaval.
Warning: if USB is connected, the system will automatically reboot after shutdown!

Params:
none
Returns:
none

reboot()

sys.reboot()

Reboot system.
In Lua shell Ctrl+D can be also used to reboot.
Short pres on power button can be also used to reboot.

Params:
none
Returns:
none

wkupint([wkup_int])

res = sys.wkupint([wkup_int])

Set or get wakeup interval.
Wake up interval can be set to enable automatic wakeup in regular intervals.
Params:
wkup_int optional; wakeup interval in minutes (values > 0 are accepted)
no parameter: return current value
Returns:
res current or new wakeup interval in minutes

schedule(val)

sys.schedule(val)

Schedule next wake up or alarm.

Params:
    val   wakeup or alarm time
        0:    wakeup or alarm on next wakeup interval
        > 0:   wakeup or alarm after 'val' seconds
        table: wakeup or alarm on specific time, table format:
            {year=yyyy, month=mm, day=dd, hour=hh, min=mn, sec=ss}
Returns:
    none   (logs info if enabled)

onshutdown(cb_func)

sys.onshutdown(cb_func)

Set callback function to be executed before shutdown.
If called without parameter, disables the callback.

Params:
    cb_func   lua function to be executed on shutdown, prototype
        function cb_func(res)
            res   integer, shutdown reason
Returns:
    none

onreboot(cb_func)

sys.onreboot(cb_func)

Set callback function to be executed before reboot.
If called without parameter, disables the callback.
Params:
    cb_func   lua function to be executed on reboot, prototype
        function cb_func(res)
            res   integer, reboot reason
Returns:
    none

onalarm(cb_func)

sys.onalarm(cb_func)

Set callback function to be executed on RTC alarm.
If called without parameter, disables the callback.

Params:
    cb_func   lua function to be executed on RTC alarm, prototype
        function cb_func(res)
            res   integer, always 0
Returns:
    none

onkey(cb_func)

sys.onkey(cb_func)

Set callback function to be executed on powr ker UP or DOWN.
If called without parameter, disables the callback.

Warning: LONG press (> 2 sec) will shutdown/reboot the system!

Params:
cb_func lua function to be executed on power key up/down, prototype
function cb_func(res)
res integer; 1: key UP, 2: key down
Returns:
none

retarget(stdio_id)

res = sys.retarget(stdio_id)

Change stdio (input/output device). All input and output will be redirected to the new device.
Params:
stdio_id id of the new device
0 redirect to **usb serial port** (*/dev/ttyACM0 on Linux*)
1 redirect to **hw UART port**
2 redirect to **bluetooth SPP** (must be configured)
Returns:
res: true on success, false on error

tick()

tick = sys.tick()

Returns time elapsed from system (RTC) start in micro seconds.

Params:
none
Returns:
res time from system start in micro seconds

elapsed(from_time)

interval = sys.elapsed(from_time)

Returns time elapsed from earlier time (usually from *sys.tick()* function) in micro seconds.

Params:
from_time: earlier time in micro seconds
Returns:
res: elapsed time in micro seconds

random([maxval] [,minval])

rnd= sys.random(from_time)

Returns random number. Optional limits can be set.

Params:
    minval   optional; mamal random number to return
    maxval   optional; maximal random number to return
Returns:
    rnd   random number

get_params()

par = sys.get_params()

Get current system parameters in string representation.
System parameters is special Lua table **__SYSPAR** which can contain the parameters used on system boot. The table can be saved to the special user flash area (not visible externally) and is read on boot.

Params:
    nil
Returns:
    par   string representation of __SYSPAR table read from system flash area
           negative number on error

get_sysvars()

c_heap, wdg = sys.get_sysvars()

Get current system variables in string representation.

System variables are special Lua variables which are set on system boot. Thay are saved to user flash area (not visible externally) and are read and set on boot.
For now only C heap size and watchdog timeout value are used

Params:
    nil
Returns:
    c_heap   C heap size read from system flash
    wdg   wdg timeout value read from system flash

save_params()

sys.save_params()

Save system parameters and system variables to user flash.
System parameters is special Lua table **__SYSPAR** which can contain the parameters used on system boot. The table can be saved to the special user flash area (not visible externally) and is read on boot.
System variables are special Lua variables which are set on system boot. Thay are saved to user flash area (not visible externally) and are read and set on boot.
For now only C heap size and watchdog timeout value are used

Params:
    nil
Returns:
    par   string representation of __SYSPAR table read from system flash area
           negative number on error

gpio module

GPIO	Function in gpio module	Voltage (V)	Connector			
			11	35	6(0.1”)	Breakout
0	IO, EINT0, UART3_RX (*)	2.8	-	-	-	-
1	IO, EINT1, ADC13, UART3_TX, CTP_SCL	2.8		3		D1
2	IO, EINT2, PWM0, ADC11, CTP_SDA	2.8		2		E1
3	IO, PWM1, ADC12	2.8		5		B1
18	IO, EINT13	2.8, 3.3	5,7		4	
13	IO, EINT11, PWM0	2.8, 3.3	6		5	
46	IO, EINT20	1.8		1		D6
30	IO, EINT16	2.8		25		
27	IO, SPI_SCK	2.8		4		C1
28	IO, SPI_MOSI	2.8		8		E2
29	IO, SPI_MISO	2.8		7		A1
43	IO, I2C_SCL	2.8, 3.3	3,9	30	2	B6
44	IO, I2C_SDA	2.8, 3.3	4,8	32	1	B5
10	IO, UART1_Rx	2.8		33		A5
11	IO, UART1_TX	2.8		34		A6
17	IO, RED LED	2.8				
15	IO, GREEN LED	2.8				
12	IO, BLUE LED	2.8				
19	IO, PWM1	2.8		31		D5
47	IO, TFT LSCK0	1.8		19		D4
48	IO, TFT LSDA0	1.8		21		B4
49	IO, TFT LSA0	1.8		22		A4
50	IO, TFT LPTE, EINT22	1.8		20		C4
38	IO, TFT LSRSTB	1.8		24		
39	IO, TFT LSCE_B	1.8		23		
52	I, EINT23, CTP_EINT	2.8		35		

(*) ADC, Battery voltage

mode(pin, mode)

gpio.mode(pin, mode)

Set the operating mode for selected GPIO pin.

Params:
- pin: GPIO pin number, see GPIO table for available pins
- mode: pin mode, use global constants:
  INPUT, OUTPUT, INPUT_PULLUP, INPUT_PULLDOWN

Returns:
- none, error if not valid pin or mode

write(pin, level)

gpio.write(pin, level)

Set the pin output to HIGH (1) or LOW (0). Pin mode must be set to output.

Params:
- pin: GPIO pin number, see GPIO table for available pins
- level: pin level, use global constants: HIGH or LOW

Returns:
- none, error if not valid pin or mode

toggle(pin)

gpio.toggle(pin)

Toggle the pin output HIGH -> LOW or LOW -> HIGH. Pin mode must be set to output.

Params:
- pin: GPIO pin number, see GPIO table for available pins

Returns:
- none, error if not valid pin or mode

read(pin)

state = gpio.read(pin)

Set the pin output to HIGH (1) or LOW (0). Pin mode must be set to output.
Params:
- pin: GPIO pin number, see GPIO table for available pins

Returns:
- state: pin state: 0 or 1
  error if not valid pin or mode

pwm_start(pin)

gpio.pwm_start(pin)

Configure selected GPIO pin for PWM operation.

Params:
pin: GPIO pin number, see GPIO table for available pins

Returns:
none, error if PWM mode not available on pin

pwm_stop(pin)

gpio.pwm_stop(pin)

Stop PWM on selected pin.

Params:
pin: GPIO pin number, see GPIO table for available pins

Returns:
none, error if pin not opened for PWM

pwm_clock(pin, clksrc, div)

gpio.pwm_clock(pin, clksrc, div)

Set the main PWM clock source.
Main PWM clock (pwm_clk) is set to 13000000 / div or 32768 / div !!
Params:
pin: GPIO pin number, see GPIO table for available pins
clksrc: PWM clock source: 0 -> 13MHz; 1 -> 32.768 kHz
div: division 0->1, 1->2, 2->4, 3->8

Returns:
none, error if pin not opened for PWM

pwm_count(pin, count, tresh)

gpio.pwm_count(pin, count, tresh)

Set PWM in count mode.
PWM FREQUENCY is: pwm_clk / count

Params:
pin: GPIO pin number, see GPIO table for available pins
count: the pwm cycle: 0 ~ 8191
tresh: treshold: value at which pwm gpio goes to LOW state: 0 ~ count

Returns:
none, error if pin not opened for PWM

pwm_freq(pin, freq, duty)

gpio.pwm_freq(pin, freq, duty)

Set PWM in frequency mode.
PWM FREQUENCY is: freq

Params:
pin: GPIO pin number, see GPIO table for available pins
freq: the pwm frequency in Hz: 0 ~ pwm_clk
duty: PWM duty cycle: 0 ~ 100

Returns:
none, error if pin not opened for PWM

eint_open(pin, [tpar])

res = gpio.eint_open(pin, [tpar])

Configure selected GPIO pin for external interrupt (EINT) operation.
Not all parameters have to be present in tpar, is some parameter is missing, default value is used.
Note: use gpio.mode() to configure the pin as input and if pullup/pulldown is used.
Params:
pin: GPIO pin number, see GPIO table for available pins
tpar: optional; Lua table with eint parametersmode(pin, mode)
autounmask: 1: unmask after callback; default 0
autopol: 1: auto change polarity after callback; default 0
sensitivity: 0: level sesitive; 1: edge sensitive; default 1
polarity: 0: high->low trigger; 1: low->high trigger; default 0
deboun: 1: enable HW debounce, 0: disable it; default 1
debountime: HW debounce time in msec; default 10
count: if >0, callback function will be executed after 'count' interrupts
Returns:
res: 0 if OK, negative number on error

eint_close(pin)

res = gpio.eint_close(pin)

Close selected GPIO pin as external interrupt (EINT) pin.

Params:
pin: GPIO pin number, see GPIO table for available pins

Returns:
res: 0 if OK, negative number on error

eint_mask(pin, mask)

res = gpio.eint_mask(pin, mask)

Mask selected GPIO pin EINT.

If *autounmask* option is not set, next interrupt must be enabled in callback function.

Params:
pin: GPIO pin number, see GPIO table for available pins
mask: 0: mask (disable) EINT operation; 1: unmask (enable) EINT operation
Returns:
res: mask value if OK, negative number on error

eint_on(cb_func)

gpio.eint_on(cb_func)

Set Lua callback function to be executed on external interrupt (EINT).
If called without parameter, disables the callback.
Params:
cb_func lua function to be executed on EINT, prototype
function cb_func(pin, value, count, time)
pin integer, pin number on which interrupt occurred
value pin level
count total number of interrupts
time time elapsed from last interrupt in micro seconds
Returns:
none

adc_config(chan, [period, count])

res = gpio.adc_config(chan, [period, count])

Configure selected ADC channel pin for ADC operation.
ADC channel must be configured before start function can be used.
Available channels are:
0: Battery voltage
1: ADC value on GPIO-1 (ADC15)
2: ADC value on GPIO-2 (ADC13)
3: ADC value on GPIO-3

Params:
chan: adc channel
period: optional; measurement period in msec; default 5 msec
Count: optional; how many measurement to take before issuing the result,
time between measurements is 'period'; default 1
time between results is 'period' * 'count'
Returns:
res: 0 if OK, negative number if error

adc_start(chan, [repeat], [cb_func])

res = gpio.adc_start(chan, [repeat], [cb_func])

Start ADC measurement on selected channel and return result if no callback function is given..

Params:
chan: adc channel configured with gpio.adc_configure
repeat: optional; repeat the measurement 'repeat' times;
1: measure only once
>1000: continuous measurement
default 1; only valid when callback function is given
cb_func: optional; Lua callback function to be executed on adc result
function cb_func(ival, fval, chan)
ival integer ADC value
fval float ADC result
chan channel on which the measurement is taken

Returns:
res: negative number if error
float ADC result if no callback function is given in V
0 if callback function given and no error

adc_stop(chan)

res = gpio.adc_stop(chan)

Stop ADC measurement on selected channel if the channel was configured for continuous/repeat measurement.

Params:
chan: adc channel configured with gpio.adc_configure

Returns:
res: 0 if ok, negative number if error
2 channel was not configured for repeat measurement

i2c module

I2c supports hardware i2c on GPIO43&GPIO44.
3.3V interface is available on 0.1” pins and 11pin connector with pullup resistors (10K) included. On 35pin connector, the pins are 2.8V and pullup resistors must be externally provided.

setup(addr [,speed])

speed = i2c.setup(addr [,speed])

Configures i2c interface. Slave address is 7-bit.
Fast mode or high speed mode is automatically selected based on ‘speed’ argument.
For speed <= 400 fast mode is selected, for speed > 400 high speed mode is selected.
Maximum speed is 6500 kHz, minimum 12 kHz.

Params:
- addr: slave device 7-bit address
- speed: optional; transfer speed i kHz, 12~6500; default 100

Returns:
- speed: actual transfer speed, negative number on error

write(data1 [,data2] [,dataN])

res = i2c.write(data1 [,data1] [,dataN])

Send data to i2c slave.
Data can be given as 8-bit number, table of 8-bit numbers or string.
Up to 10K bytes can be sent at once.

Params:
- data: data to be sent to i2c device; number, lua table or lua string

Returns:
- res: number of bytes sent to device, negative number on error

read(size [,format])

res = i2c.read(size [,format])

Receive data from i2c slave.
Data can be received to lua string, string of hex values or lua table.
Up to 10K bytes can be received at once.

Params:
size: number of bytes to receive, 1 ~ 10240
format: optional; if not given, data are received to lua string
"*h" receive to string of hex values separated by ";"
"*t" receive to lua table

Returns:
res: lua string or lua table containing received data
nil on error

txrx(data1 [,data2] [,dataN], size [,format])

res = i2c.txrx(data1 [,data1] [,dataN], size [,format])

Send data to i2c slave and receive data.
Data can be given as 8-bit number, table of 8-bit numbers or string.
Maximum of 8 bytes can be sent, up to 10240 bytes can be received.

Params:
data: data to be sent to i2c device; number, lua table or lua string, max 8 bytes
size: number of bytes to receive, 1 ~ 10240
format: optional; if not given, data are received to lua string
"*h" receive to string of hex values separated by ";"
"*t" receive to lua table

Returns:
res: lua string or lua table containing received data
nil on error

close()

res = i2c.close()

Close i2c interface.

Params:
nil

Returns:
res: status; 0 if OK, negative number on error

spi module

spi module supports hardware spi on GPIO27-29.
SPI pins are available only on 35pin connector (or expansion board) as 2.8V interface.
Use **level shifters** if connecting to higher voltage device!
CS output pin can be defined in setup, if defined it is automatically activated (set LOW) during the transfer.
Additional output pin (DC) can also be defined. If defined, it is used in some functions.

setup([config])

res = spi.setup([config])

Configures SPI interface.
Configuration options are given in form of lua table.

Params:
config: lua table containing configuration options in form *option=value*
- mode — spi transfer mode, 0 ~ 3; default 0
- endian — endianess; 0: little; 1: big; default 0
- msb — bit transfer mode; 0: send LSB first; 1: send MSB first; default 1
- speed — SPI transfer speed in kHz; 125 ~ 16000; default 6400
- cs — CS (chip sellect) output gpio pin; default: not used
- dc — DC output gpio pin; default: not used

Returns:
res: result: 0 if OK, negative number on error

write(data1 [,data2] [,dataN])

res = spi.write(data1 [,data1] [,dataN])

Send data to SPI device.
Data can be given as 8-bit number, table of 8-bit numbers or string.
Maximum 16384 bytes can be sent.

Params:
data: data to be sent to spi device; number (0~255), lua table or lua string
Special strings can be used for control:
- "*c" — send first byte as command, DC (if defined) will be set LOW before first byte transfer and reset to HIGH after the first byte is transferred
- "*s" — CS will not be deactivated (set to HIGH) after transfer

Returns:
res: number of bytes sent to device, negative number on error

read(size [,control])

res, data = spi.read(size [control])

Receive data from SPI slave.

Data can be received to lua string, string of hex values or lua table.
Maximum 16384 bytes can be received.

Params:

size:	number of bytes to receive, 1 ~ 16384	
control:	optional;	more than one control strings can be given
		if none of *"*h"*, *"*t"* is present, data are received to lua string
	"*h"	receive to string of hex values separated by ";"
	"*t"	receive to lua table
	"*s"	CS will not be deactivated (set to HIGH) after transfer

Returns:

res:	number of bytes read from device, negative number on error
data:	lua string or lua table containing received data

txrx(data1 [,data2] [,dataN] [,control], size)

bsent, brecv, data = spi.txrx(data1 [,data2] [,dataN] [,control], size)

Send data to SPI slave and receive data in same transaction.
Data can be given as 8-bit number, table of 8-bit numbers or string.
Maximum of 16384 bytes can be sent and received.

Params:

data:	data to be sent to SPI device; number (0~255), lua table or lua string	
size:	number of bytes to receive after sending	
control:	optional;	more than one control strings can be given
		if none of *"*h"*, *"*t"* is present, data are received to lua string
	"*h"	receive to string of hex values separated by ";"
	"*t"	receive to lua table
	"*s"	CS will not be deactivated (set to HIGH) after transfer
	"*c"	send first byte as command, DC (if defined) will be set LOW before first byte transfer and reset to HIGH after the first byte is transferred
	"*w"	include in read data the data received during sending. brecv will be *size+bsent*

Returns:

bsent:	number of sent bytes, negative number on error
brecv:	number of received bytes
data:	lua string or lua table containing received data

close()

res = spi.close()

Close SPI interface.

Params:

nil

Returns:

res:	status; 0 if OK, negative number on error

net module

To get the status of the UDP or TCP connection execute `print(ref).`
To enable garbage collector to free the data used by the connection, execute `ref=nil`.
ref is the reference to tcp or udp connection obtained by *tcp_create* or *udp_create* function.
GPRS must be configured with *setapn()* function before using any of net functions.

tcp_create(host, port, cb_func, [data])

Tcp_ref = net.tcp_create(host, port, cb_func, [data])

Creates TCP connection and connects to 'host' on 'port'.
'host' can be IP address or domain name.
If string 'data' is given, it will be sent to host after connection.

Params:
- host: host IP or domain name
- port: integer, tcp port to connect to (1 ~ 65535)
- cb_func: Lua callback function, prototype:
  function cb_func(tcp_ref, event)
  - tcp_ref: tcp connection
  - event: tcp event which caused the call:
    1: tcp is connected, can send data
    2: more date can be sent
    3: data ready for read
    4: pipe broken, disconnected
    5: host not found, not connected
    6: connection closed
- data: optional; string data to send after connection

Returns:
- tcp_ref: reference to tcp connection to be used in other function

tcp_connect(tcp_ref, host, port, [data])

res = net.tcp_connect(tcp_ref, host, port, [data])

Connects to already created tcp connection. If tcp connection is connected, it is disconnected first. 'host' can be IP address or domain name.
If string 'data' is given, it will be sent to host after connection.
Params:
- tcp_ref: tcp reference obtained with *net.tcp_create()*
- host: host IP or domain name
- port: integer, tcp port to connect to (1 ~ 65535)
- data: optional; string data to send after connection

Returns:
- res: 0 if OK, negative number if error

tcp_write(tcp_ref, data)

res = net.tcp_write(tcp_ref, data)

Send data to tcp connection. Tcp connection must in connected state.

Params:
- tcp_ref: tcp reference obtained with *net.tcp_create()*
- data: string data to send

Returns:
- res: 0 if OK, negative number if error

tcp_read(tcp_ref, size)

res, data = net.tcp_read(tcp_ref, size)

Read data from tcp connection.
This function can be used from callback function on read event.

Params:
- tcp_ref: tcp reference obtained with *net.tcp_create()*
- size: maximum size of data to read

Returns:
- res: size or read data, negative number if error
- data: string, read data; nil if error

udp_create(port, cb_func)

udp_ref = net.udp_create(port, cb_func)

Creates UDP connection on local port 'port'. No connection is made.
Params:
- port: integer, local port (1 ~ 65535)
- cb_func: Lua callback function, prototype:
  function cb_func(udp_ref, event)
  - udp_ref: udp connection
  - event: udp event which caused the call:
    - 2: more date can be sent
    - 3: data ready for read
    - 4: pipe broken, disconnected
    - 6: connection closed
- data: optional; string data to send after connection

Returns:
- udp_ref: reference to udp connection to be used in other function

udp_write(udp_ref, host, port, data)

res = net.udp_write(tcp_ref, host, port, data)

Connect to 'host' on UDP port 'port' and send data using udp connection 'udp_ref'.
Response will be handled by callback function.

Params:
- udp_ref: udp reference obtained with *net.udp_create()*
- host: host IP or domain name
- port: integer, udp port to connect to (1 ~ 65535)
- data: string data to send

Returns:
- res: 0 if OK, negative number if error

udp_read(udp_ref, size)

res, data = net.udp_read(udp_ref, size)

Read data from udp connection. This function can be used from callback function on read event.

Params:
- udp_ref: udp reference obtained with *net.udp_create()*
- size: maximum size of data to read

Returns:
- res: size or read data, negative number if error
- data: string, read data; nil if error

close(ref)

res = net.close(ref)

Close TCP or UDP connection. TCP connection will be disconnected if connected.
To enable garbage collector to free the data used by the connection, execute `ref=nil`, where *ref* if the reference to tcp or udp connection obtained by *tcp_create* or *udp_create* function.

Params:
- ref: udp | tcp reference obtained with *net.tcp_create()* or *net.udp_create()*

Returns:
- res: 0 if OK, negative number if error

ntptime(tz, [cb_func])

net.ntptime(tz, [cb_func])

Update **RTC date-time** from ntp server.

The function runs in background until it gets the time from ntp server or timeout (60 sec) expires. If callback function is given, it is executed after the time is set or on error.

If no callback function is given, debug info is printed.

Params:
tz: time zone, *-12 <= tz <= 14*
cb_func: optional; Lua callback function, prototype:
function cb_func(res)
res integer, 0 if time updated, -1 on error
Returns:
none

setapn(ref)

res = net.setapn(apn_par)

Configure GPRS APN.
GPRS connection parameters should be obtained from mobile provider.

Params:
apn_par: Lua table with APN parameters:
apn: GPRS provider APN
useproxy: optional; proxy needed for connection;
1 use proxy; 0 do not useproxy; default 0
Used only if ***useproxy=1***:
proxy: proxy IP or domain name
proxyport: proxy port (1 ~ 65535)
proxytype: optional; the type of the proxy connection; default 0
0: The '*not specified*' type
1: The WSP, Connection less type
2: The WSP, Connection oriented type
3: The WSP, Connection less, security mode type
4: The WSP, Connection oriented, security mode type
5: The WTA, Connection less, security mode type
6: The WTA type, Connection oriented, security mode type
7: The HTTP type
8: The HTTP - enable TLS type
9: The STARTTLS type
proxyuser: optional; proxy user name; default “”
proxypass: optional; proxy password; default “”

Returns:
res: 0 if OK, negative number if error

https module

This module supports the tcp communication using **http** or **https** protocol.
It can be used to communicate with http server with or without SSL.
Use URL which starts with *http://....* to connect without SSL.
GPRS must be configured with *setapn()* function before using any of https functions.

get(url [,fname | buf_size] [,wait_time])

res = https.get(url [,fname | buf_size] [,wait_time])

len, data = https.get(url [,fname | buf_size] [,wait_time])

Connect to http(s) server and get the response.
The response is handled by **Lua callback function** which has to be set using ***https.on()*** function before using this function.
If **no callback function** is registered, the function will wait for the response.
The response can be received to buffer or to the file.
If receiving to buffer, maximum buffer length can be specified to limit the amount of data received.
If the 2nd (optional) parameter is given and it is of the **string** type, response will be saved to file.
If the 2nd (optional) parameter is given and it is of the **integer** type, response will be saved to buffer of that maximum size.
If the 3rd (optional) parameter is given and no response callback function is registered, the function will wait for the response for maximum of that time.

See https.on() function for details on response handling;

Params:

url:	string; server url, can contain parameters	
fname:	optional;	if given, the response will be saved to file with the name 'fname'
buf_size:	optional;	maximum buffer size: 1K ~ 128K; default: 16K
wait_time:	optional;	maximum time in seconds to wait for the response if no callback function is registered; default: 30 sec

Returns:

res:	0 if OK, negative number on error
len:	received data length if no callback function is registered, negative number on error
data:	string, received data if no callback function is registered, nil on error

post(url, post_data [,fname] [,wait_time])

res = https.post(url, post_data [,fname | buf_size] [,wait_time])

len, data = https.post(url, post_data [,fname | buf_size] [,wait_time])

Connect to http(s) server, post data and get the response.
The response is handled by **Lua callback function** which has to be set using ***https.on()*** function before using this function.
If **no callback function** is registered, the function will wait for the response.
The response can be received to buffer or to the file.
If receiving to buffer, maximum buffer length can be specified to limit the amount of data received.

If the 3rd (optional) parameter is given and it is of the **string** type, response will be saved to file.
If the 3rd (optional) parameter is given and it is of the **integer** type, response will be saved to buffer of that maximum size.
If the 4th (optional) parameter is given and no response callback function is registered, the function will wait for the response for maximum of that time.

If *post_data* is **string**, the data is posted as `Content-Type: application/x-www-form-urlencoded`
This way you can post json string.
If *post_data* is Lua **table,** the data is posted as *multipart* data. The table must contain parameters in the form *param_name = value*, where *value* can be string or number.
If *param_name* is "**file**", then the file with name given in *value* is posted.
Only **one** file can be included in post data.

See https.on() function for details on response handling.

Params:

url:	string; server url, can contain parameters	
post_data:	string or Lua table containing post data	
fname:	optional;	if given, the response will be saved to file with the name 'fname'
buf_size:	optional;	maximum buffer size: 1K ~ 128K; default: 16K
buf_size:	optional;	maximum buffer size: 1K ~ 128K; default: 16K
wait_time:	optional;	maximum time in seconds to wait for the response

Returns:

res:	0 if OK, negative number on error

cancel()

https.cancel()

Cancel http(s) unfinished function.
Can be used if previous **get** or **post** function resulted in unclosed connection.

Params:
nil

Returns:
nil

getstate()

state = https.getstate()

Get the currenh http(s) state.
New **get** or **post** function can be executed only if the state is **0**.

Params:
Nil

Returns:

state:	https state:
	0: idle, ready for new request
	1: receiving to buffer
	2: receiving to file

on(method [,cb_func])

res = https.on(method [, cb_func])

Register Lua callback function which will be executed on received data event.
If no cb_func is given, existing (if any) callback function is unregistered.

Params:

method: string; event on which the function will be executed:
"header" execute function when response header is received
"response" execute function when response data are received

cb_func: optional; Lua callback function

header prototype:

function cb_func(hdr)

hdr string; received response header

response prototype:

function cb_func(state, data, len, more)

state receive status:
1: data received to buffer
3: data received to buffer and contains non printable characters probably binary data
2: data received to file
-1: receiving to string failed
-2: receiving to file failed

data string; received response data
"__Receive_to_File__" if receiving to file

len length of the received data

more if receiving to buffer, length of the data which could not be saved to buffer
If receiving to file, negative number on error

Returns:

res: status:
-1: error, noting changed
0: callback unregistered
1: callback registered

Callback function example

Here is an example of using “response” callback function to process response from server:

```
function onresp(state, data, len, more)
    if (state == 1) or (state == 3) then
        print("Received to buffer")
        if more > 0 then
            print(("Partial data received: %d of %d"):format(len, len+more))
        else
            print("Data received, length: "..len)
        end
        if (len < 3000) and (state == 1) then
            print(data)
        elseif state == 3 then
            print ("Data contains non printable characters")
        end
    elseif state == 2 then
        print("Received to file")
        if more < 0 then
            print("Error receiving to file, received len: "..len)
        else
            print("File saved successfuly, len: "..len)
        end
    else
        print(("Error: state=%d, len=%d, more=%d"):format(state, len, more))
    end
end

https.on(“response”, onresp)

https.get(“http://loboris.eu/rephone/test.php?data=123)
```

Blocking function call example

Here is an example of using get function in blocking mode (function will wait for the response):

```
-- unregister callback function, if already registered
https.on(“response”)

len, data = https.get(“http://loboris.eu/rephone/test.php?data=123)

print(data)
```

Note: http://loboris.eu/rephone/test.php can be used to test GET & POST functions.

email module

This module supports sending email.
If port parameter is 25, the function uses regular smtp protocol, otherwise SSL connection is used.
GPRS must be configured with *setapn()* function before using any of email functions.

send(param)

res = email.send(param)

Connect to SNTP server and send email to recipient.

Params:

param:	Lua table containing email parameters	
	"host"	SMTP server domain or IP address
	"port"	SMTP server connection port
	"user"	optional; user name
	"pass"	optional; user password
	"to"	recipient's email address
	"from"	sender's email
	"from_name"	optional; sender's name; default: same as "from"
	"subject"	optional if *msg* is given; email subject
	"msg"	optional if *subject* is given; email message body

Returns:

res:	0 if OK, negative number on error

Example:

```
eml = {
    host="smtp.gmail.com",
    port=465,
    user="mygmail@gmail.com",
    pass="my_gmail_password",
    to="recipient@gmail.com",
    from="mygmail@gmail.com",
    from_name="rephone",
    subject="Test",
    msg="test email message from RePhone" }

res = email.send(eml)
```

ftp module

This module supports FTP communication. Uses PASV mode for communication.
GPRS must be configured with *setapn()* function before using any of ftp functions.

connect(param)

res = ftp.connect(param)

Connect to ftp server and log in.

Params:
- param: Lua table containing ftp connection parameters
  - "host" FTP server domain or IP address
  - "port" optional; FTP server connection port; default: 21
  - "user" user name
  - "pass" user password

Returns:
- res: 0 if OK, negative number on error

disconnect()

res = ftp.disconnect()

Log out and disonnect from ftp server.

Params:
- nil

Returns:
- res: 0 if OK, negative number on error

list(file [,option])

len, slist = ftp.list(file)

res = ftp.list(file, file_name])

nlin, tlist = ftp.list(file, "*totable")

List files on current directory of the connected ftp server.

Params:
- file: file specification (“*”, “*.lua”, data/*.*”, etc)
- option: optional;
  - not given: the result is returned as Lua string
  - “file_name”: the result is written to the file “file_name”
  - “*totable”: the result is returned as Lua table

Returns:
- res: 0 if save to file OK, negative number on error
- tlist: Lua table containing the listing
- slist: Lua string containing the listing
- len: string length if listing returned as string, negative number on error
- nlin: number of items in table, negative number on error

recv(remote_file [,local_file])

res = ftp.recv(remote_file, local_file)

len, str_file = ftp.recv(file, “*tostring”)

Receive remote file from ftp server to local file or string.

Params:
- remote_file: remote file name to receive
- local_file: local file name
  If “*tostring” the result is returned as Lua string

Returns:
- res: 0 if save to file OK, negative number on error
- str_file: Lua string containing the remote file
- len: the length of the file received to string, negative number on error

send(file_name [,remote_file] [,“*append”])

res = ftp.send(file_name)

res = ftp.send(string, remote_file [,”*append”])

res = ftp.send(file_name, “*append”)

res = ftp.send(file_name, remote_file)

res = ftp.send(file_name, remote_file, “*append”)

Send local file or string to ftp server.

Params:

file_name:	local file name or string to send If file with that name exists on local file system, the file is sent, otherwise the content of this parameter is sent and *remote_file* parameter is mandatory
remote_file:	optional; remote file name
"*append"	optional; append the file or string to remote file, if not given overwrite remote file

Returns:

res:	0 if OK, negative number on error

chdir(remote_dir)

res = ftp.chdir(remote_dir)

Set current directory on ftp server.

Params:

remote_dir:	new remote directory to set as current

Returns:

res:	0 if save to file OK, negative number on error

getdir(remote_dir)

len, res = ftp.chdir()

Get current directory on ftp server.

Params:

nil

Returns:

res:	string; ftp server remote dir
len:	length of res, negative number on error

mqtt module

MQTT originally stood for MQ Telemetry Transport, but is now just known as "MQTT". It is a publish/subscribe, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimize network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. These principles also turn out to make the protocol ideal of the emerging “machine-to-machine” (M2M) or “Internet of Things” world of connected devices, and for mobile applications where bandwidth and battery power are at a premium.

This module supports MQTT communication. Multiple mqtt connections are supported.
GPRS must be configured with setapn() function before using any of mqtt functions.

create(config)

mqtt_id = mqtt.create(config)

Create and configure new mqtt client connection. Returns mqtt connection id which is used in other functions.
No connection is established.
Client status can be checked with *print(mqtt_id)*.
To destroy the client and free memory set mqtt_id to **nil**.

Params:

config: Lua table containing mqtt client connection parameters

“host”	MQTT broker domain or IP address
“port”	optional; MQTT broker connection port; default: 1883
“user”	optional; user name
“pass”	optional; user password, mandatory if user name given
“clientid”	optional; MQTT Client ID, default: “*RephoneMQTTClient*”
“qos”	optional; QoS (Quality of Service); default 0
“onmessage”	optional; Lua callback function to be executed on new message

```
function cb_func(len, topic, msg)
    len     received message length
    topic   string; message topic
    msg     string; received message
```

“ondisconnect”	optional; Lua callback function to be executed on disconnect

```
function cb_func(host)
    host    the host from which the client was disconnected
```

Returns:

mqtt_id: mqtt client id to be used in other mqtt functions or **nil** on error

connect(mqtt_id [,check_int] [,ka_int])

res = mqtt.connect(mqtt_id [,check_int] [,ka_int])

Connect to mqtt broker, set check for message and keep alive intervals.

Params:

mqtt_id	mqtt id obtained with *create* function
check_int	optional; interval in seconds to check for messages; default 30 `5 <= check_int <= 300 &  check_int <= (ka_int/2)`
ka_int	optional; keep alive interval in seconds; default 60 `30 <= ka_int <= 600`

Returns:

res:	0 if connected; negative number on error

disconnect(mqtt_id)

res = mqtt.disconnect(mqtt_id)

Disconnect from mqtt broker.

Params:

mqtt_id	mqtt id obtained with *create* function

Returns:

res:	0 if disconnected; negative number on error

addtopic(mqtt_id, topic [,qos])

mqtt.addtopic(mqtt_id ,topic [,qos])

Set mqtt client topic. Up to 5 topics can be added per client.

Params:

mqtt_id	mqtt id obtained with *create* function
topic	string; topic name, max length 31
qos	optional; topic's QoS; default: client's QoS

Returns:

nil

subscribe(mqtt_id)

mqtt.subscribe(mqtt_id)

Subscribe to topics added with *addtopic()* function.

Params:

mqtt_id	mqtt id obtained with *create* function

Returns:

res:	0 if subscribed; negative number on error

unsubscribe(mqtt_id [,topic])

mqtt.unsubscribe(mqtt_id [,topic])

Unsubscribe from topics added with *addtopic()* function or from all topics.

Params:

mqtt_id	mqtt id obtained with *create* function
topic	optional; topic name to unsubscribe from; if none given, unsubscribe from all topics

Returns:

res:	0 if unsubscribed; negative number on error

publish(mqtt_id, topic, message [,qos])

mqtt.publish(mqtt_id ,topic, message [,qos])

Publish to mqtt topic.

Params:

mqtt_id	mqtt id obtained with *create* function
topic	string; topic name, max length 31
message	string; message to publish
qos	optional; topic's QoS; default: client's QoS

Returns:

res:	0 if OK; negative number on error

sms module

Functions for manipulating SMS message.
Maximum sms message length (send and receive) is 640 bytes.

send(num, msg [,cb_func])

res = sms.send(num, msg [,cb_func])

Send sms message to gsm number.
Wait for message to be sent if no cb_func is given.

Params:
- num: string; gsm phone number to which to send the message
- msg: string; message body
- cb_func: optional; Lua callback function to be executed after message is sent
  - function cb_func(stat)
    - stat 0 if OK, negative number on error

Returns:
- res: 0 if OK, negative number on error

numrec([box_type])

res = sms.numrec([box_type])

Check the number of received sms messages.

Params:
- box_txpe: optional; message box type, default 1
  - 0x01 Inbox.
  - 0x02 Outbox.
  - 0x04 Daft box.
  - 0x08 Unsent box. Messages to be sent.
  - 0x10 SIM card.
  - 0x20 Archive box.

Returns:
- res: number of messages, negative number on error

list([msg_state] [cb_func])

res = sms.list([msg_state] [cb_func])

Get the list of available message id's.

Params:
- msg_state: optional; message state, default 1
  - 0x01 Unread.
  - 0x02 Read.
  - 0x04 Sent.
  - 0x08 Unsent (to be sent).
  - 0x10 Draft.

cb_func: optional; Lua callback function to be executed after message list ready
function cb_func(tlist)
tlist Lua table containing message id's
which can be used to read or delete the message

Returns:
res: *if cb_func is not given:*
Lua table containing message id's which can be used to read or delete the message
if cb_func is not given:
Message list query result; 0: OK

read(msg_id [,cb_func])

time, msg = sms.read(msg_id)

res = sms.read(msg_id ,cb_func)

Read the message at index msg_id.
Message indexes can be obtained with sms.list() function.

Params:
msg_id: message index from which to read the message
cb_func: optional; Lua callback function to be executed message is read
function cb_func(time, msg)
time Lua table containing message sent time
Table keys are: `sec,min,hour,day,month,year`
msg Lua string, the message body

Returns:
res: 0 if OK, negative number on error
time: string; message sent time
msg: string; message body

delete(msg_id [,cb_func])

res = sms.delete(msg_id [,cb_func])

Delete sms message at index msg_id.
Message indexes can be obtained with sms.list() function.
Waits for message to be deleted if no cb_func is given.

Params:
msg_id: message index from which to delete
cb_func: optional; Lua callback function to be executed after message is deleted
function cb_func(stat)
stat 0 if OK, negative number on error

Returns:
res: 0 if OK, negative number on error

onmessage([cb_func])

res = sms.onmessage([cb_func])

Set Lua callback function to be executed when new message arrives.
If called without parameter, removes exiting callback function reference.

Params:
cb_func: optional; Lua callback function to be executed after new message arrives
function cb_func(msg_id, gsm_num, msg)
msg_id massage id
gsm_num string; phone number from which message is received
msg string; message body
Returns:
res: 0 if OK, negative number on error

sim_info()

stat, imei, imsi = sms.siminfo()

Returns SIM card status, IMEI and IMSI numbers.

Params:
nil

Returns:
stat: SIM card status:

```
-2  No active SIM card detected.
-1  Failed to get status.
 0  No SIM card is detected or the SIM card is not working.
 1  The SIM card is working.
```

imei: IMEI number, returned only if stat=1
imsi: IMSI number, returned only if stat=1

timer module

Timers are used to execute the Lua function on regular intervals. Multiple timers can be created (up to 10).
After the timer is created it can be stopped, paused, restarted, resumed and deleted.

Timer can be in one of the following states:

- ***running*** *timer is running, callback function is executed on regular intervals*
- ***paused*** *timer is running, but the callback function is not executed*
- ***stopped*** *timer is stopped, callback function remains registered, timer can be restarted*
- ***deleted*** *timer is stopped, callback function unregistered, cannot be restarted*

Minimal timer interval is 5 msec.
To check the timer's state and some statistics use *print(timer_id)*.
Be careful when using small interval timers, it can affect performance.

create(interval, cb_func [,state])

timer_id = timer.create(interval, cb_func [,state])

Create new timer. Returns timer id which is used in other timer functions.
Timer can be created in running, stopped or paused state,
Timer status can be checked with *print(timer_id)*.
To destroy the timer and free memory set timer_id to **nil**.

Params:

interval:	timer interval in msec
cb_func:	Lua callback function to be executed on timer interval
	function cb_func(timer_id)
	timer_id id of the timer for which the function is called
state:	optional; timer timer state after creation; default: 0
	0: running
	-1: running, execute callback function immediately after creation
	1: paused
	2: stopped

Returns:

timer_id:	timer id to be used in other timer functions

delete(timer_id)

res = timer.delete(timer_id)

Stop and delete the timer.
To destroy the timer and free memory set timer_id to **nil**.

Params:
    timer_id: timer id obtained with create function

Returns:
    res: 0 if OK, negative number on error

pause(timer_id)

timer.pause(timer_id)

Pause the timer. Timer is running but callback function is not executed while paused.

Params:
    timer_id: timer id obtained with create function
Returns:
    nil

resume(timer_id [,sync])

timer.resume(timer_id [,sync])

Resume paused timer. Callback function will be executed on timer interval.

Params:
    timer_id: timer id obtained with create function
    sync: optional; if 1, waits the next timer interval before resuming; default: 0, do not wait
Returns:
    nil

stop(timer_id)

timer.stop(timer_id)

Stop the timer. Timer is not but callback function is preserved.

Params:
    timer_id: timer id obtained with create function
Returns:
    nil

start(timer_id)

timer.start(timer_id)

Start stopped timer. Callback function is executed.

Params:
    timer_id: timer id obtained with create function
Returns:
    nil

changecb(timer_id, cb_func)

timer.changecb(timer_id, cb_func)

Change timer's callback function.
Timer is paused first, callback function is changed, timer is resumed.

Params:
    timer_id: timer id obtained with create function
    cb_func: new Lua callback function
Returns:
    nil

changeint(timer_id, int)

timer.changeint(timer_id, int)

Change timer's interval function.
Timer is stopped first, interval is changed, timer is started.

Params:
    timer_id: timer id obtained with create function
    int: new timer's interval
Returns:
    nil

getid(timer_id)

id = timer.getid(timer_id)

Returns timer's internal id.

Params:
    timer_id: timer id obtained with create function
Returns:
    id: timer's ID

getstate(timer_id)

state = timer.getstate(timer_id)

Returns timer's current state. See create function for state v

Params:
    timer_id: timer id obtained with create function
Returns:
    state: timer's state:
                 0: running
                 1: paused
                 2: stopped
                 3: deleted

bt module

Bluetooth module.
Support communication via Bluetooth using SPP profile.
Lua interactive shell can be **redirected** to bluetooth SPP interface after client is connected.
While redirection is active, client can send the string "*<Return2ShEll>*" to redirect Lua shell to USB serial again.

start(name)

res = bt.start(name)

Turns on BT and starts bluetooth connection manager if not already started.
Makes BT visible under the name '*name*'.
Host name and BT MAC address are reported id debugg logging is enabled.

Params:
    name: rephone BT device name
Returns:
    res: 0 if OK, negative number if error

spp_start([cb_func])

res = bt.spp_start([cb_func])

Starts bluetooth SPP profile and registers *onreceive* callback function if given.
BT mus be started using *bt.start()* function.
onconnect & *ondisconnect* callback functions can be registered separately.
No spp security is used, clients can connect to the device without PIN code.

Params:
    cb_func optional; Lua callback function to be executed on data receive
        function cb_func(addr, len, data)
            addr string; client's BT MAC
            len integer; data string length
            data string data received from connected BT client
Returns:
    res: 0 if OK, negative number if error

spp_write(data)

res = bt.spp_write(data)

Writes string to bluetooth spp connection.
If redirection to BT is active, this function does nothing.
Params:
    data: string to be sent to BT connection
Returns:
    res: number of written bytes if OK, negative number if error

spp_stop()

res = bt.spp_stop()

Disconnets client if connected and stops SPP profile.

Params:
    none
Returns:
    none

stop()

res = bt.stop()

Disconnets client if connected and stops SPP profile if active.
Stops bluetooth connection manager and turns off BT.

Params:
    none
Returns:
    none

onconnect([cb_func])

bt.onconnect([cb_func])

Register Lua callback function which will be executed on client connect.
If no cb_func is given, existing (if any) callback function is unregistered.

Params:
    cb_func: optional; Lua callback function, prototype:
        function cb_func(addr, stat)
            addr   string; client's BT MAC
            stat   integer; connection status
Returns:
    nil

ondisconnect([cb_func])

bt.ondisconnect([cb_func])

Register Lua callback function which will be executed on client disconnect.
If no cb_func is given, existing (if any) callback function is unregistered.

Params:
    cb_func: optional; Lua callback function, prototype:
        function cb_func(addr)
            addr   string; client's BT MAC
Returns:
    nil

uart module

Communication over serial port.
USB serial port (/dev/ttyACM0, MTK Debug port) and **hardware UART** are supported.
Only the UART which is not used as Lua Shell input/output can be created.

create(port, cb_func [,param])

uart_id = uart.create(port, cb_func [,param])

Create new uart. Returns uart id which is used in other uart functions.

Params:
- port: 0: create uart on USB serial port; 1: create uart on hw UART port
- cb_func: Lua callback function to be executed on data received event
  - function cb_func(uart_id, len, data)
    - uart_id: id of the uart for which the function is called
    - len: length of the received data
    - data: string; received data
- param: optional; Lua table containing communication parameters; default; 115200,8,1,n
  Possible table key values are:
  - bit: number of data bits: 5 ~ 8
  - par: parity: 0 ~ 4; **0**=none; **1**=odd; **2**=even; **3**=mark; **4**=space
  - stop: nuber of stop bits: 1 ~ 3; **1**=1; **2**=2; **3**=1.5
  - bdr: baud rate: 75 ~ 921600;
    only limits are checked, enter the correct value!

Returns:
- uart_id: uart id to be used in other timer functions

delete(uart_id)

uart.delete(uart_id)

Deletes the uart. Unregister callback function

Params:
- uart_id: uart ID obtained with uart.create() function

Returns:
- nil

write(uart_id, data)

uart.write(uart_id, data)

Write data to uart.

Params:
- uart_id: uart ID obtained with uart.create() function
- data: string; data to write

Returns:
- nil

term module

Term module enables advanced Lua shell terminal communication if used with ANSI/VT100 compatible terminals. Screen positioning functions are included, as well as full featured file editor and ymodem file transfer.

Clear functions

term.clrscr()

Clears the screen, position the cursor at (1,1).

term.clreol()

Clears the the current line from current x position to the end of line.

Cursor move functions

term.moveto(x,y)

Move the cursor to given position.

term.moveup(), term.moveup(), term.moveleft(), term.moveright()

Move the cursor up, down, left or right.

getlines()

lin = term.getlines()

Get current number of terminal lines.

Params:
    nil
Returns:
    lin: number of terminal lines

getcols()

col = term.getcols()

Get current number of terminal columns.

Params:
    nil
Returns:
    col: number of terminal columns

setlines(lin)

term.setlines(lin)

Set the number of terminal lines.

Params:
    lin: number of terminal lines
Returns:
    nil

setcols(col)

term.setcols(col)

Set the number of terminal columns.

Params:
    col: number of terminal columns
Returns:
    nil

getcx()

x = term.getcx()

Get current character's X position.

Params:
    nil
Returns:
    x: current X position

getcy()

y = term.getcy()

Get current character's Y position.

Params:
    nil
Returns:
    y: current Y position

getchar([wait])

c = term.getchar([wait])

Get character from terminal input buffer or wait for one.

Some special key characters can be received, they are converted to the following Lua constants which can be used for comparison:
UP, DOWN, LEFT, RIGHT, HOME, END, PAGEUP, PAGEDOWN, ENTER, TAB, BACKSPACE, ESC, DEL, INS, CTRL_Z, CTRL_A, CTRL_C, CTRL_E, CTRL_T, CTRL_U, CTRL_K, CTRL_L, UNKNOWN

When used all constants must be prefixed by module name, for example: term.HOME.

Params:
wait: if wait=1, waits for the character,
otherwise returns the character from buffer if there is one
Returns:
c: ASCII code of the received character

getstr(x, y, maxlen [,outstr])

str = term.getstr(x, y, maxlen [,outstr])

Move cursor to position x,y; optionally print the given string and wait for input.

The input is terminated when Enter or Ctrl-C is pressed.

Simple line editing is possible, left&right arrow keys, Home, End, Backspace and Del keys can be used.

Params:
x: X screen position
y: Y screen position
maxlen: maximum input length, 1~254
outstr: optional; string to print before input, max 128 characters

Returns:
str: received string, if **Ctrl-C** was pressed, **nil** value is returned;

edit(file)

term.edit(file)

Edit the text file.
If the file with given name does not exist, it is created.

The editing area is determined by current number of lines and columns.

Arrow keys, Home, End, Backspace, Del, PgUp, PgDown keys can be used to navigate through file.
Insert key can be used to switch between insert/overwrite mode.
ab inserts 4 spaces.
Ctrl-C exits editing without saving the file.
Ctrl_Z exits editing and, if file was changed, prompts for saving.

When the file is read, all tab characters are converted to 4 spaces!

While editing file, no Lua callback function can be executed, be careful if editing for long time.

Params:
file: file name

Returns:
none

yrecv([file_name])

term.yrecv([file_name])

Receives the file over serial line using ymodem protocol.
Text and binary files can be received.

Params:
- file_name: optional; save file under that name; default: save under original name

Returns:
- nil: prints result on terminal

ysend(file_name [,host_fname])

term.ysend(file_name [,host_fname])

Send the file over serial line using ymodem protocol.
Text and binary files can be sent.

Params:
- file_name: file to send
- host_fname: optional; send with different name; default: send with original name

Returns:
- nil: prints result on terminal

sensor module

Support for various sensors.

At the moment the following sensors are supported:

- **DHT-11/DHT-22**
- Bosh Sensortec **BME280** Temperature, pressure & humidity sensor in I2C mode
- **DS18B20, DS18S20, DS1822 and DS28EA00**
  DS1820 type sensors can be connected in "parasite power" mode
  in which only **DATA**& **GND** pins are used. See data sheet for details.

dht_get(pin, type)

t, h, stat = sensor.dht_get(pin, type)

Read temperature and humidity from DHT-11/DHT22 sensor.

Params:
- pin: GPIO pin to which the sensor data pin is connected
- type: sensor type: **0**: DHT-11, **1**: DHT-22

Returns:
- t: temperature
- h: humidity
- stat: 0 if successful, negative number on error

ds_init(pin)

res = sensor.ds_init(pin)

Initiate 1-wire interface on selected pin and test if any device is connected.

Params:
- pin: GPIO pin to which the DS sensor data pin is connected

Returns:
- res: 1 if at least one sensor is detected; 0 if error or no device connected

ds_search()

n = sensor.ds_search()

Returns number of DS1820 devices detected on 1-wire bus.

Params:
- nil

Returns:
- n: number of DS1820 type devices detected

ds_setres(n, res)

rres = sensor.ds_setres(n, res)

Set desired measurement resolution, 9, 10, 11 or 12 bit.
Higher resolution requires longer measurement time. See data sheet for details.

Params:
- n: sensor device number: 1 to number of sensors detected with *sensor.ds_search*
- res: resolution: 9~12 bit

Returns:
- rres: actual resolution set or negative number on error

ds_getres(n)

res, mtime = sensor.ds_getres(n)

Get current resolution and expected measurement time.

Params:
- n: sensor device number: 1 to number of sensors detected with *sensor.ds_search*

Returns:
- res: curent sensor's resolution
- mtime: expected measurement time in msec

ds_gettemp(n)

t, stat = sensor.ds_gettemp(n)

Start measurement, wait for measurement result and return it.

Params:
- n: sensor device number: 1 to number of sensors detected with *sensor.ds_search*

Returns:
- t: measured temperature in degree C, *-9999* on error
- stat: measurement status: **0** if measurement OK, error code on error

ds_startm(n)

res = sensor.ds_startm()

Start temperature measurement on all connected devices. Does not wait for result

Params:
- n: sensor device number: 1 to number of sensors detected with *sensor.ds_search*

Returns:
- res: **0** if OK, **-1** on error (no devices connected)

ds_get(n)

t, stat = sensor.ds_gettemp(n)

Get the last temperature value. Measurement must be started with *sensor.ds_startm()*.

Params:
n: sensor device number: 1 to number of sensors detected with *sensor.ds_search*

Returns:
t: measured temperature in degree C, *-9999* on error
stat: measurement status: **0** if measurement OK, error code on error

ds_getrom(n)

rom = sensor.ds_getrom(n)

Get the ROM values (unique 8-byted sensor identification) to Lua table.

Params:
n: sensor device number: 1 to number of sensors detected with *sensor.ds_search*

Returns:
rom: Lua table containing 8 values of the sensor ID

bme280_init(addr [,mode [,per]])

res = sensor.bme280_init(addr [,mode [,per]])

Initiate BME280 sensor on I2C address. I2C interface will be initiated automatically

Params:
addr: sensor's i2c addres; **0x76** or **0x77**
mode: optional; operating mode; **0**, **1** or **3**; default: 0
0: sleep mode, no operation, lowest power
1: forced mode, perform one measurement, return to sleep mode
3: normal mode, continuous measurements with inactive periods between
per: optional; standby period between measurements in normal mode in msec; default: 125

Returns:
res: 0 if OK, negative number on error

bme280_getmode()

mod, per = sensor.bme280_getmode()

Get current BME280 sensor operating mode

Params:
nil

Returns:
mode: current operating mode(0, 1, 3) if OK, negative number on error
per: current standby period in msec if OK, nil on error

bme280_setmode(mode [,per])

res = sensor.bme280_setmode(mode [,per])

Set BME280 sensor operating mode

Params:
- mode: operating mode; **0**, **1** or **3**; default: 0
  - **0**: sleep mode, no operation, lowest power
  - **1**: forced mode, perform one measurement, return to sleep mode
  - **3**: normal mode, continuous measurements with inactive periods between
- per: optional; standby period between measurements in normal mode in msec; default: previous value

Returns:
- res: 0 if OK, negative number on error

bme280_get()

t, h, p = sensor.bme280_get()

Get measurement results.
In normal mode, last measurement results are returned, otherwise new measurement is initiated and results returned.

Params:
- nil

Returns:
- t: temperature, float value in °C
- h: humidity, float value in %
- p: pressure, float value in Pa (Pascal)

audio module

Support for playing and recording audio.
Works with **Xadow audio** board, but should work with speaker and microphone connected via expansion board too.

play(file [,format])

res = audio.play(file)

Start playing audio from file.

Params:
- file: name of the audio file
- format: optional; audio format; default MP3;
  the system will usually recognize the format without explicitly giving one

```
NONE = -1
AMR = 3
MP3 = 5,
AAC = 6
WAV = 13
MIDI = 17
IMELODY = 18
OTHER = 100
```

Returns:
- res: 0 if OK, negative number on error

pause()

res = audio.pause()

Pause current playback.

Params:
- nil

Returns:
- res: 0 if OK, negative number on error

resume()

res = audio.resume()

Resume paused playback.

Params:
- nil

Returns:
- res: 0 if OK, negative number on error

stop()

res = audio.stop()

Stop current playback.

Params:
    nil

Returns:
    res: 0 if OK, negative number on error

get_time()

t = audio.get_time()

Get current playback time.

Params:
    nil

Returns:
    t: playback time in msec, 0 if not playing

set_volume(vol)

audio.set_volume(vol)

Set playback volume.

Params:
    Vol: playback volume: 0 ~ 6

Returns:
    nil

get_volume()

vol = audio.get_volume()

Get current playback volume.

Params:
    nil

Returns:
    vol: current playback volume

record(file [,format])

res = audio.record(file)

Start recording audio to file.

If the file already exists, error code -3 will be returned. Delete first iv want to overwrite.

Params:
- file: name of the audio file
- format: optional; audio format; default AMR;
  - AMR = 3
  - WAV = 13

Returns:
- res: 0 if OK, negative number on error

rec_pause()

res = audio.rec_pause()

Pause current recording.

Params:
- nil

Returns:
- res: 0 if OK, negative number on error

rec_resume()

res = audio.rec_resume()

Resume paused recording.

Params:
- nil

Returns:
- res: 0 if OK, negative number on error

rec_stop()

res = audio.rec_stop()

Stop current recording.

Params:
- nil

Returns:
- res: 0 if OK, negative number on error

bit module

Bitwise operations on numbers.

More in separate document ***BitOp.pdf***.

json module

Lua Cjson module.

More in separate document ***Lua CJSON 2.1.0 Manual.pdf***.

struct module

Module for converting data to and from C structs.

More in separate document ***Lua struct module.pdf***.

lcd module

Module for displaying data on TFT lcd display modules with touchscreen support.

More in separate document ***LCD module.pdf***.