

# Lua for RePhone (Xadow GSM+BLE)

## Programming Manual



Ver.: LuaRephone 1.0.2  
LoBo 08/2016

# table of contents

table of contents.....	1
os module.....	5
copy(from_file, to_file).....	5
mkdir(name).....	5
rmdir(name).....	5
exists(name).....	5
list(filespec).....	5
compile(name).....	6
shell_linetype(type).....	6
table2str(tbl [,list]).....	6
sys module.....	7
sys.ver().....	7
mem().....	7
battery().....	7
ledblink([led_id]).....	7
usb().....	8
wdg([wdg_tmo]).....	8
c_heapsize([size]).....	8
noacttime([noact_tmo]).....	8
shutdown().....	9
reboot().....	9
wkupint([wkup_int]).....	9
schedule(val).....	9
onshutdown(cb_func).....	10
onreboot(cb_func).....	10
onalarm(cb_func).....	10
onkey(cb_func).....	10
retarget(stdio_id).....	11
tick().....	11
elapsed(from_time).....	11
random([maxval] [,minval]).....	11
get_params().....	12
get_sysvars().....	12
save_params().....	12
gpio module.....	14
mode(pin, mode).....	15
write(pin, level).....	15
toggle(pin).....	15
read(pin).....	15
pwm_start(pin).....	15
pwm_stop(pin).....	16
pwm_clock(pin, clksrc, div).....	16
pwm_count(pin, count, tresh).....	16
pwm_freq(pin, freq, duty).....	16
eint_open(pin, [tpar]).....	17
eint_close(pin).....	17
eint_mask(pin, mask).....	17
eint_on(cb_func).....	18
adc_config(chan, [period, count]).....	18

adc_start(chan, [repeat], [cb_func]).....	18
adc_stop(chan).....	19
i2c module.....	20
setup(addr [,speed]).....	20
write(data1 [,data2] [,dataN]).....	20
read(size [,format]).....	20
txrx(data1 [,data2] [,dataN], size [,format]).....	21
close().....	21
spi module.....	22
setup([config]).....	22
write(data1 [,data2] [,dataN]).....	22
read(size [,control]).....	22
txrx(data1 [,data2] [,dataN] [,control], size).....	23
close().....	23
net module.....	24
tcp_create(host, port, cb_func, [data]).....	24
tcp_connect(tcp_ref, host, port, [data]).....	24
tcp_write(tcp_ref, data).....	24
tcp_read(tcp_ref, size).....	25
udp_create(port, cb_func).....	25
udp_write(udp_ref, host, port, data).....	25
udp_read(udp_ref, size).....	26
close(ref).....	26
ntptime(tz, [cb_func]).....	26
setapn(ref).....	27
https module.....	28
get(url).....	28
post(url).....	28
on(method [,cb_func]).....	28
cancel().....	29
email module.....	30
send(param).....	30
ftp module.....	31
connect(param).....	31
disconnect().....	31
list(file [,option]).....	31
recv(remote_file [,local_file]).....	32
send(file_name [,remote_file] [,“*append”]).....	32
chdir(remote_dir).....	33
getdir(remote_dir).....	33
mqtt module.....	34
create(config).....	34
connect(mqtt_id [,check_int] [,ka_int]).....	34
disconnect(mqtt_id).....	35
addtopic(mqtt_id, topic [,qos]).....	35
subscribe(mqtt_id).....	35
unsubscribe(mqtt_id [,topic]).....	35
publish(mqtt_id, topic, message [,qos]).....	36
sms module.....	37
send(num, msg [,cb_func]).....	37
numrec([box_type]).....	37
list([msg_state] [cb_func]).....	37
read(msg_id [,cb_func]).....	38

delete(msg_id [,cb_func]).....	38
onmessage([cb_func]).....	38
sim_info().....	39
timer module.....	40
create(interval, cb_func [,state]).....	40
delete(timer_id).....	40
pause(timer_id).....	41
resume(timer_id [,sync]).....	41
stop(timer_id).....	41
start(timer_id).....	41
changeCb(timer_id, cb_func).....	41
changeint(timer_id, int).....	42
getId(timer_id).....	42
getState(timer_id).....	42
bt module.....	43
start(name).....	43
spp_start([cb_func]).....	43
spp_write(data).....	43
spp_stop().....	44
stop().....	44
onconnect([cb_func]).....	44
ondisconnect([cb_func]).....	44
uart module.....	46
create(port, cb_func [,param]).....	46
delete(uart_id).....	46
write(uart_id, data).....	46
term module.....	47
Clear functions.....	47
Cursor move functions.....	47
getlines().....	47
getcols().....	47
setlines(lin).....	47
setcols(col).....	48
getcx().....	48
getcy().....	48
getchar([wait]).....	48
getstr(x, y, maxlen [,outstr]).....	49
edit(file).....	49
yrecv([file_name]).....	49
ysend(file_name [,host_fname]).....	49
sensor module.....	51
dht_get(pin, type).....	51
ds_init(pin).....	51
ds_search().....	51
ds_setres(n, res).....	51
ds_getres(n).....	52
ds_gettemp(n).....	52
ds_startm(n).....	52
ds_get(n).....	52
ds_getrom(n).....	53
bit module.....	54
json module.....	54
struct module.....	54

lcd module.....	54
-----------------	----

# 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, the **simple** input type is used where the user can only input characters, use Backspace key to delete last char and accept the line with Enter.

**Advanced** input type can be 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.

To use the advanced type, used terminal emulator must have 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(tbl [,list])

### tstr = os.table2str(tbl)

### os.table2str(tbl,1)

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

Params:

tbl: Lua table

list: **optional**; if 1, list table content on terminal

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 size 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), the system is shutdown after no activity

If called without parameters, the current no activity timeout is returned.

```
noact_tmo  optional;    > 0  set no activity timeout in seconds
              0         reset no activity timeout
no parameter: return current value
```

```
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 interval.

**Warning: if USB is connected, the system will automatically reboot after shutdown!**

none

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.

none

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.

wkup\_int    **optional**; wakeup interval in minutes (values > 0 are accepted)  
no parameter: return current value

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 <code>function cb_func(res)</code>
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 <code>function cb_func(res)</code>
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 <code>function cb_func(res)</code>
res	integer, always 0

Returns:

none

## onkey(cb\_func)

### sys.onkey(cb\_func)

Set callback function to be executed on power 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; minimal 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. They 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. They 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, if 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 parameters  
**autounmask**: 1: unmask after callback; default 0  
**autopol**: 1: auto change polarity after callback; default 0  
**sensitivity**: 0: level sensitive; 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  
**debouncetime**: 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

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*

<b>mode</b>	spi transfer mode, 0 ~ 3; default 0
<b>endian</b>	endianess; 0: little; 1: big; default 0
<b>msb</b>	bit transfer mode; 0: send LSB first; 1: send MSB first; default 1
<b>speed</b>	SPI transfer speed in kHz; 125 ~ 16000; default 6400
<b>cs</b>	CS (chip select) output gpio pin; default: not used
<b>dc</b>	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: <code>function cb_func(tcp_ref, event)</code> tcp_ref      tcp connection event        tcp event which caused the call: 1: tcp is connected, can send data 2: more data 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 <code>net.tcp_create()</code>
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 data 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 (30 sec) expires. If callback function is given, it is executed after the time is set or error. If no callback function is given, debug info is printed.

Params:

tz: time zone,  $-12 \leq tz \leq 14$   
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 can 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 protocol.

It can be used to communicate with http server with or without SSL.

GPRS must be configured with *setapn()* function before using any of https functions.

## get(url)

### res = https.get(url)

Connect to http(s) server and get the response.

The response is handled by Lua callback function. If no callback function is given, the response is printed to standard output. See *https.on()* for details.

Params:

url: string; server url, can contain parameters

Returns:

res: 0 if OK, negative number on error

## post(url)

### res = https.post(url, post\_data)

Connect to http(s) server, post data and get the response.

The response is handled by Lua callback function. If no callback function is given, the response is printed to standard output. See *https.on()* for details.

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.

Params:

url: string; server url, can contain parameters

post\_data: string or Lua table containing post data; tag\_https\_response\_cb\_ref

Returns:

res: 0 if OK, negative number on error

## on(method [,cb\_func])

### https.on(method [, cb\_func])

Declare Lua callback function which will be executed on received data event.

If no *cb\_func* is given, existing (if any) callback function is unreferenced.

Params:

method: event on which the function will be executed  
"header" execute function when response header is received  
"response" execute function when response data is received  
cb\_func: optional; Lua callback function, prototype:  
function cb\_func(hdr) (for header method)  
hdr string; received response header  
function cb\_func(data, more) (for response method)  
data string; received response data  
more 1: more data will follow; 0: all data received

Returns:

nil

## cancel()

### https.cancel()

Cancel http(s) unfinished function.

Params:

nil

Returns:

nil

# 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 SMTP 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 <i>msg</i> is given; email subject
"msg"	optional if <i>subject</i> 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 disconnect 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, "\*table" )**

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 <i>remote_file</i> 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
"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:sms module

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 <i>create</i> 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 <i>create</i> function
topic	string; topic name, max length 31
message	string; message to publish
qos	<i>optional</i> ; 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 Draft 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     message 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<br><code>function cb_func(timer_id)</code><br><code>timer_id</code> id of the timer for which the function is called  |
| state:    | <b>optional</b> ; timer timer state after creation; default: 0<br>0: running<br>-1: running, execute callback function immediately after creation<br>1: paused<br>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 if debug 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 must 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()**

Disconnects client if connected and stops SPP profile.

Params:

none

Returns:

none

## **stop()**

### **res = bt.stop()**

Disconnects 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 unreferenced.

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 unreferenced.

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 0,0.

### **term.cleol()**

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

## getcxc()

### 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, an empty string 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. [Tab](#) 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!

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])**

Sends 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 **DHT-11/DHT-22** and **DS18B20, DS18S20, DS1822 and DS28EA00** sensors are supported.

**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 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.

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: current 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

## bit module

Bitwise operations on numbers.

See the separate document ***BitOp.pdf***.

## json module

Lua Cjson module.

See the separate document ***Lua CJSON 2.1.0 Manual.pdf***.

## struct module

Module for converting data to and from C structs.

See the separate document ***Lua struct module.pdf***.

## lcd module

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

See the separate document ***LCD module.pdf***.