

Lua for RePhone (Xadow GSM+BLE)

Programming Manual



Ver.: LuaRephone 0.9.4 Beta
LoBo 07/2016

table of contents

table of contents.....	1
os module.....	4
copy(from_file, to_file).....	4
mkdir(name).....	4
rmdir(name).....	4
exists(name).....	4
list(filespec).....	4
compile(name).....	5
sys module.....	6
sys.ver().....	6
mem().....	6
battery().....	6
ledblink([led_id]).....	6
usb().....	7
wdg([wdg_tmo]).....	7
noacttime([noact_tmo]).....	7
shutdown().....	7
reboot().....	8
wkupint([wkup_int]).....	8
schedule(val).....	8
onshutdown(cb_func).....	8
onreboot(cb_func).....	9
onalarm(cb_func).....	9
onkey(cb_func).....	9
retarget(stdio_id).....	9
tick().....	10
elapsed(from_time).....	10
random([maxval] [,minval]).....	10
gpio module.....	11
mode(pin, mode).....	12
write(pin, level).....	12
toggle(pin).....	12
read(pin).....	12
pwm_start(pin).....	12
pwm_stop(pin).....	13
pwm_clock(pin, clksrc, div).....	13
pwm_count(pin, count, tresh).....	13
pwm_freq(pin, freq, duty).....	13
eint_open(pin, [tpar]).....	14
eint_close(pin).....	14
eint_mask(pin, mask).....	14
eint_on(cb_func).....	15
adc_config(chan, [period, count]).....	15
adc_start(chan, [repeat], [cb_func]).....	15
adc_stop(chan).....	16
i2c module.....	17
setup(addr [,speed]).....	17
write(data1 [,data2] [,dataN]).....	17
read(size [,format]).....	17

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

pause(timer_id).....	37
resume(timer_id).....	37
change_cb(timer_id, cb_func).....	38
bt module.....	39
start(name).....	39
spp_start([cb_func]).....	39
spp_write(data).....	39
spp_stop().....	40
stop().....	40
onconnect([cb_func]).....	40
ondisconnect([cb_func]).....	40

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 exists.

Params:

name: string, file name

Returns:

res: 0 if file exists, 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

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. After setting the new value, system must be rebooted to take effect. 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

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 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 interval.

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 wakeup 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 power key 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)

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 (<i>/dev/ttyACM0 on Linux</i>)
1	redirect to hw UART port
2	redirect to bluetooth SPP (must be configured)

Returns:

None

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

Params:

from_time earlier time in micro seconds

Returns:

res time from system start 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

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*

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 select) 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: <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

Timer related functions. Multiple timers can be created.
Minimal timer interval is 1 msec.
Be careful when using small interval timers, it can affect performance.

`create(interval, cb_func)`

`timer_id = timer.create(interval, cb_func)`

Create and start new timer. Returns timer id which is used in other timer functions.
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
	<code>function cb_func(timer_id)</code>
	<code>timer_id</code> id of the timer for which the function is called

Returns:

timer_id:	timer id to be used in other timer functions
-----------	--

`delete(timer_id)`

`res = timer.delete(timer_id)`

Stop and delete.
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. Callback function is not executed while paused.

Params:

timer_id:	timer id obtained with create function
-----------	--

Returns:

nil

`resume(timer_id)`

`timer.resume(timer_id)`

Resume paused 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

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`