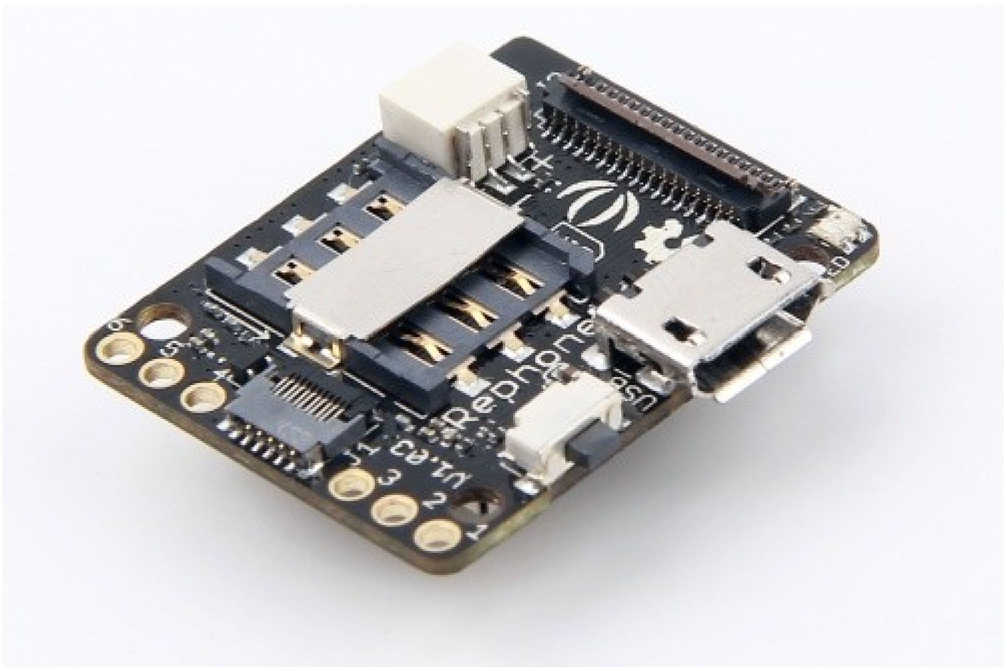
Lua for RePhone

(Xadow GSM+BLE)

Manual



Ver.: LuaRephone 0.9.2 Beta
LoBo 07/2016

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 fs 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

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 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 interaval.

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

Params:
    none
Returns:
    none

reboot()

sys.reboot()

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

Params:
    none
Returns:
    none

wkupint([wkup_int])

res = sys.wkupint([wkup_int])

Set or get wakeup interval.
Wake up interval can be set to enable automatic wakeup in regular intervals.

Params:

wkup_int	optional; wakeup interval in minutes (values > 0 are accepted) no parameter: return current value

Returns:

res	current or new wakeup interval in minutes

schedule(val)

sys.schedule(val)

Schedule next 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

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

Returns:

none

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		E2
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
25	IO, EINT15	2.8		35		

(*) ADC, Battery voltage

mode(pin, mode)

gpio.mode(pin, mode)
Set the operating mode for selected GPIO pin.

Params:

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

Returns:
none, error if not valid pin or mode

write(pin, level)

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

Params:

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

Returns:
none, error if not valid pin or mode

toggle(pin)

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

Params:

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

Returns:
none, error if not valid pin or mode

read(pin)

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

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

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

pwm_start(pin)

gpio.pwm_start(pin)
Configure selected GPIO pin for PWM operation.

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

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

pwm_stop(pin)

gpio.pwm_stop(pin)
Stop PWM on selected pin.

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

Returns:
none, error if pin not opened for PWM

pwm_clock(pin, clksrc, div)

gpio.pwm_clock(pin, clksrc, div)
Set the main PWM clock source.
Main PWM clock (pwm_clk) is set to 13000000 / div or 32768 / div !!

Params:

pin:	GPIO pin number, see GPIO table for available pins
clksrc:	PWM clock source: 0 -> 13MHz; 1 -> 32.768 kHz
div:	division 0->1, 1->2, 2->4, 3->8

Returns:

none, error if pin not opened for PWM

pwm_count(pin, count, tresh)

gpio.pwm_count(pin, count, tresh)
Set PWM in count mode.
PWM FREQUENCY is: pwm_clk / count

Params:

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

Returns:

none, error if pin not opened for PWM

pwm_freq(pin, freq, duty)

gpio.pwm_freq(pin, freq, duty)
Set PWM in frequency mode.
PWM FREQUENCY is: freq

Params:

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

Returns:

none, error if pin not opened for PWM

eint_open(pin, [tpar])

res = gpio.eint_open(pin, [tpar])
Configure selected GPIO pin for external interrupt (EINT) operation.
Not all parameters have to be present in tpar, is some parameter is missing, default value is used.

Note: use gpio.mode() to configure the pin as input and if pullup/pulldown is used.

Params:

pin:	GPIO pin number, see GPIO table for available pins	
tpar:	optional; Lua table with eint parametersmode(pin, mode)	
	autounmask:	1: unmask after callback; default 0
	autopol:	1: auto change polarity after callback; default 0
	sensitivity:	0: level sesitive; 1: edge sensitive; default 0
	polarity:	0: high->low trigger; 1: low->high trigger; default 0
	deboun:	enable HW debounce; default 0
	debountime:	HW debounce time in msec; default 10

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.

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)	
	pin	integer, pin number on which interrupt occurred
	level	pint level

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

To get the status of the UDP or TCP connection execute **print(ref).**
To enable garbage collector to free the data used by the connection, execute **ref=nil**.

ref is the reference to tcp or udp connection obtained by *tcp_create* or *udp_create* function.

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

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

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

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

Params:

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

Returns:

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

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

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

Connects to already created tcp connection. If tcp connection is connected, it is disconnected first. 'host' can be IP address or domain name.
If string 'data' is given, it will be sent to host after connection.

Params:

tcp_ref:	tcp reference obtained with *net.tcp_create()*
host:	host IP or domain name
port:	integer, tcp port to connect to (1 ~ 65535)
data:	optional; string data to send after connection

Returns:

res:	0 if OK, negative number if error

tcp_write(tcp_ref, data)

res = net.tcp_write(tcp_ref, data)

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

Params:

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

Returns:

res:	0 if OK, negative number if error

tcp_read(tcp_ref, size)

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

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

Params:

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

Returns:

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

udp_create(port, cb_func)

udp_ref = net.udp_create(port, cb_func)

Creates UDP connection on local port ‘port’. No connection is made.

Params:

port:	integer, local port (1 ~ 65535)
cb_func:	Lua callback function, prototype:
	function cb_func(udp_ref, event)
	udp_ref — udp connection
	event — udp event which caused the call:
	2: more date can be sent
	3: data ready for read
	4: pipe broken, disconnected
	6: connection closed
data:	optional; string data to send after connection

Returns:

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

udp_write(udp_ref, host, port, data)

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

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

Params:

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

Returns:

res:	0 if OK, negative number if error

udp_read(udp_ref, size)

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

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

Params:

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

Returns:

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

close(ref)

res = net.close(ref)

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

Params:

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

Returns:

res:	0 if OK, negative number if error

ntptime(tz, [cb_func])

net.ntptime(tz, [cb_func])

Update RTC date-time from ntp server.
The function runs in background until it gets the time from ntp server or timeout (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 <= tz <= 14*
cb_func:	optional; Lua callback function, prototype:
	function cb_func(res)
	res integer, 0 if time updated, -1 on error

Returns:

None

setapn(ref)

res = net.setapn(apn_par)

Configure GPRS APN.
GPRS connection parameters 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