Lua for RePhone
(Xadow GSM+BLE)

Programming Manual

table of contents

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:

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 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

onkey(cb_func)

sys.onkey(cb_func)

Set callback function to be executed on powr ker UP or DOWN.

If called without parameter, disables the callback.

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

Params:

    cb_func   lua function to be executed on power key up/down, prototype

                 function cb_func(res)

                     res   integer; 1: key UP, 2: key down

Returns:

    none

retarget(stdio_id)

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 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		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 1

    polarity: 0: high->low trigger; 1: low->high trigger; default 0

    deboun: 1: enable HW debounce, 0: disable it; default 1

    debountime: HW debounce time in msec; default 10

    count: if >0, callback function will be executed after 'count' interrupts

Returns:

res: 0 if OK, negative number on error

eint_close(pin)

res = gpio.eint_close(pin)

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

Params:

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

Returns:

res: 0 if OK, negative number on error

eint_mask(pin, mask)

res = gpio.eint_mask(pin, mask)

Mask selected GPIO pin EINT.

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

Params:

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

mask: 0: mask (disable) EINT operation; 1: unmask (enable) EINT operation

Returns:

res: mask value if OK, negative number on error

eint_on(cb_func)

gpio.eint_on(cb_func)

Set Lua callback function to be executed on external interrupt (EINT).

If called without parameter, disables the callback.

Params:

    cb_func   lua function to be executed on EINT, prototype

                    function cb_func(pin, value, count, time)

                        pin   integer, pin number on which interrupt occurred

                        value   pin level

                        count   total number of interrupts

                        time

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 during the transfer.

Additional output pin (DC) can also be defined. If defined, its state (defined by *setdc()* function) is set before transfer.

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; 86 ~ 22000; default 4400
    - **cs** CS (chip sellect) output gpio pin; default: not used
    - **dc** DC output gpio pin; default: not used

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

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

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

Send data to SPI device.

Data can be given as 8-bit number, table of 8-bit numbers or string.

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

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

read(size [,format])

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

Receive data from SPI slave.

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

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

Send data to SPI slave then receive data in same transaction.

Data can be given as 8-bit number, table of 8-bit numbers or string.

Params:
    data: data to be sent to SPI device; number, lua table or lua string
    size: number of bytes to receive
    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 = spi.close()

Close SPI interface.

Params:
    nil

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

net module

To get the status of the UDP or TCP connection execute `print(ref).`

To enable garbage collector to free the data used by the connection, execute `ref=nil`.

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

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

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

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

Creates TCP connection and connects to 'host' on 'port'.

'host' can be IP address or domain name.

If string 'data' is given, it will be sent to host after connection.

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

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

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

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

Connects to already created tcp connection. If tcp connection is connected, it is disconnected first. 'host' can be IP address or domain name.

If string 'data' is given, it will be sent to host after connection.

Params:

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

Returns:

res:	0 if OK, negative number if error

tcp_write(tcp_ref, data)

res = net.tcp_write(tcp_ref, data)

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

Params:

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

Returns:

res:	0 if OK, negative number if error

tcp_read(tcp_ref, size)

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

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

Params:

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

Returns:

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

udp_create(port, cb_func)

udp_ref = net.udp_create(port, cb_func)

Creates UDP connection on local port 'port'. No connection is made.

Params:

port:	integer, local port (1 ~ 65535)
cb_func:	Lua callback function, prototype:

> function cb_func(udp_ref, event)
>> udp_ref   udp connection
>>
>> event   udp event which caused the call:
>>> 2:   more date can be sent
>>>
>>> 3:   data ready for read
>>>
>>> 4:   pipe broken, disconnected
>>>
>>> 6:   connection closed

data:	optional; string data to send after connection

Returns:

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

udp_write(udp_ref, host, port, data)

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

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

Params:

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

Returns:

res:	0 if OK, negative number if error

udp_read(udp_ref, size)

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

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

Params:

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

Returns:

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

close(ref)

res = net.close(ref)

Close TCP or UDP connection. TCP connection will be disconnected if connected.

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

Params:

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

Returns:

res:	0 if OK, negative number if error

ntptime(tz, [cb_func])

net.ntptime(tz, [cb_func])

Update RTC date-time from ntp server.

The function runs in background until it gets the time from ntp server or timeout (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

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 datag_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 fallow; 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 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:

    param: Lua table containing email parameters

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

Returns:

    res: 0 if OK, negative number on error

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

res = email.send(eml)
```

ftp module

This module supports FTP communication. Uses PASV mode for communication.

.

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

connect(param)

res = ftp.connect(param)

Connect to ftp server and log in.

Params:

    param: Lua table containing ftp connection parameters

            "host"   FTP server domain or IP address

            "port"   optional; FTP server connection port; default: 21

            "user"   user name

            "pass"   user password

Returns:

    res: 0 if OK, negative number on error

disconnect()

res = ftp.disconnect()

Log out and disonnect from ftp server.

Params:

    nil

Returns:

    res: 0 if OK, negative number on error

list(file [,option])

len, slist = ftp.list(file)

res = ftp.list(file, file_name])

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

List files on current directory of the connected ftp server.

Params:

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

Returns:

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

recv(remote_file [,local_file])

res = ftp.recv(remote_file, local_file)

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

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

Params:

remote_file:	remote file name to receive
local_file:	local file name
	If "*tostring" the result is returned as Lua string

Returns:

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

send(file_name [,remote_file] [,"*append"])

res = ftp.send(file_name)

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

res = ftp.send(file_name, "*append")

res = ftp.send(file_name, remote_file)

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

Send local file or string to ftp server.

Params:

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

Returns:

res:	0 if OK, negative number on error

`chdir(remote_dir)`

res = ftp.chdir(remote_dir)

Set current directory on ftp server.

Params:

remote_dir:	new remote directory to set as current

Returns:

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

`getdir(remote_dir)`

len, res = ftp.chdir()

Get current directory on ftp server.

Params:

nil

Returns:

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