Converter LUA API Documentation V1.1

API Functions Description

api.**ledControl**(*color*, *state*)

> This function turns on or off the on board LED. It can be used for debugging purposes.
>
> > ***Arguments***
> > *color* (integer) - The color of LED to control: 0 for *red*, 1 for *green* and 2 for *blue*
> > *state* (integer) - The new state of the LED: 1 for *on* and 0 for *off*
>
> *example:*
> ```
> api.ledControl(2,1) --turns on blue LED
> ```

api.**delayms**(*ms*)

> This function makes the execution be paused for *ms* milliseconds.
>
> > ***Arguments***
> > *ms* (integer) - The number of milliseconds to delay
>
> *example:*
> ```
> api.delayms(1000) --delay one second
> ```

status, *port*, *answer* = api.**loraSend**(*ack*, *timeout*, *msg*)

> This function sends buffer *msg* to LoRa. Acknowledged or non-acknowledged transport can be used using *ack* parameter. Maximum execution time is limited by *timeout* miliseconds.
>
> > ***Arguments***
> > *ack* (integer) - Selects acknowledged (1) or non-acknowledged (0) transport mode
> > *timeout* (integer) - The maximum execution time in milliseconds, used in acknowledged mode
> > *msg* (string) - String to be sent to LoRa
> > ***Returns***
> > *status* (integer) - Positive or zero for success, negative for failure
> > *port* (integer) - Nil or port on which the answer was received
> > *answer* (string) - Nil or non-zero length string containing gateway answer

> *example:*
> ```
> --sends 0xCCBBAA35 to LoRa with 20s timeout and acknowledged mode
> msg = pack.pack('<b4', 0xCC, 0xBB, 0xAA, 0x35)
> status,port,answer = api.loraSend(1,20000, msg)
> ```

status, *answer* = api.**nbSend**(*addr*, *port*, *msg*, *timeout*, *flag*)

> This function sends buffer *msg* to NB. Maximum length of Rx and Tx messages is 512 Bytes. Maximum execution time is limited by *timeout* miliseconds.
>
> > ***Arguments***
> > *addr* (string) - IP address
> > *port* (integer) - Port
> > *msg* (string) - String to be sent to NB
> > *timeout* (integer) - The maximum execution time in milliseconds
> > *flag* (integer, optional) - Optional flag, specifies the type of message transmission: *0x100* is Exception Message: Send message with high priority, *0x200* is Release Indicator: indicate release after next message, *0x400* is Release Indicator: indicate release after next message has been replied to. More details can be found in:
> > https://www.quectel.com/UploadImage/Downlad/Quectel_BC95_AT_Commands_Manual_V1.9.pdf
> > ***Returns***
> > *status* (integer) - Zero for success, negative for failure
> > *answer* (string) - Nil or non-zero length string containing the answer
>
> *example:*
> ```
> -- sends "test message" string to IP 185.8.239.192 on port 5566 with 6s timeout
> status,answer = api.nbSend("185.8.239.192", 5566, "test message", 6000)
> ```

status, *c*, *a*, *ci*, *answer* = api.**mbusTransaction**(*msg*, **timeout**, **retry=1**)

> This function transmits *msg* and waits *timeout* milliseconds for the answer. The transmission is retried *retry* times. The *status* contains information about communication status and *c*, *a*, *ci* and *answer* contains MBus answer data. Turn on MBus using **mbusState** first.
>
> > ***Arguments***
> > *msg* (string) - Message to send to MBus
> > *timeout* (integer, optional) - The maximum time in milliseconds to wait for MBus device answer
> > *retry* (integer, optional) - Optional number of retransmissions, defaults to 1
> > ***Returns***
> > *status* (integer) - Number of bytes received, zero on failure
> > *c* (integer) - MBus c frame field
> > *a* (integer) - MBus a frame field
> > *ci* (integer) - MBus ci frame field
> > *answer* (string) - MBus frame payload received from the bus

> *example:*
> ```
> --sends MBus frame [0x10, 0x50, 0x30, 0x16], waits 5s for answer twice
> msg = pack.pack('<b4', 0x10, 0x50, 0x30, 0x16)
> status,c,a,ci,ans = api.mbusTransaction(msg, 5000, 2)
> ```

api.**mbusSetup**(**baudrate**, **parity**, **stopBits**, **dataBits**)

> This function configures the MBus communication interface. By default, the configuration from GUI is used, but this can be overriden using this API. Turn on MBus using **mbusState** after setting up MBus parameters using this function.
>
> > ***Arguments***
> > *baudrate* (integer, optional) - Baudrate to use for communication (up to 921600 baud)
> > *parity* (integer, optional) - Parity, 0 for none, 1 for odd and 2 for even parity
> > *stopBits* (integer, optional) - Number of stop bits, 1 or 2 allowed
> > *dataBits* (integer, optional) - Number of data bits, 7 or 8 allowed
>
> *example:*
> ```
> --setup MBus interface to 9600 Baud, 8E2
> api.mbusSetup(9600, 2, 2, 8)
> ```

api.**mbusState**(*state*)

> This function turns on the MBus circuitry, must be used before **mbusTransaction**.
>
> > ***Arguments***
> > *state* (integer) - New state of MBus circuitry: 0 for off, 1 for on (~30s power-up)
>
> *example:*
> ```
> api.mbusState(1) --turn on MBus
> ```

api.**rs485Send**(*msg*)

> This function sends *msg* to RS485 bus. Turn on RS485 using **rs485State** first.
>
> > ***Arguments***
> > *msg* (string) - Data to be sent to RS485 bus
>
> *example:*
> ```
> api.rs485Send('test') --sends 'test' string to RS485
> ```

api.**rs485Setup**(**baudrate**, **parity**, **stopBits**, **dataBits**)

> This function changes the configuration of RS485 interface
>
> > ***Arguments***
> > *baudrate* (integer, optional) - Baudrate to use for communication (up to 921600 baud)
> > *parity* (integer, optional) - Parity, 0 for none, 1 for odd and 2 for even parity

> > *stopBits* (integer, optional) - Number of stop bits, 1 or 2 allowed
> > *dataBits* (integer, optional) - Number of data bits, 7 or 8 allowed
> > *example:*
> > ```
> > api.rs485Setup(9600, 2, 1, 8) --9600 baud, 8E1
> > ```

api.**rs485State**(*state*)

> This function turns on the RS485 circuitry, must be used before **rs485Send** or **rs485Receive**.
>
> > ***Arguments***
> > *state* (integer) - New state of RS485 circuitry: 0 for off, 1 for on (fast power-up)
>
> *example:*
> ```
> api.rs485State(0) --turn off RS485
> ```

answer,*len* = api.**rs485Receive**(*msg*, *timeout*, *retry=1*)

> This function waits *timeout* milliseconds for data reception from RS485 bus. Turn on RS485 using **rs485State** first.
>
> > ***Arguments***
> > *timeout* (integer) - The maximum time in milliseconds to wait for RS485 device answer
> > ***Returns***
> > *answer* (string) - Data received from RS485 bus in given time
> > *len* (integer) - Number of bytes received
>
> *example:*
> ```
> --waits 1s for answer from RS485 bus
> ans,len = api.rs485Receive(1000)
> ```

crc = api.**modbusCrc**(*msg*)

> This function calculates Modbus request checksum.
>
> > ***Arguments***
> > *msg* (string) - Modbus request
> > ***Returns***
> > *crc* (string) - Modbus crc for request
>
> *example:*
> ```
> --calculate checksum for Modbus request 110100010002
> crc = api.modbusCrc("110100010002") --crc = "EE9B"
> ```

tick = api.**getTick**( )

> This function returns current number of milliseconds since startup. Counts up to 2^32 and then restarts from 0.
>
> > ***Returns***
> > *tick* (integer) - Number of milliseconds since startup of the device
>
> *example:*
>
> ```
> --get a timestamp, can be used for timing
> timestamp = api.getTick()
> ```

data,*error*,*acked*,*wake*,*intArg* = api.**getGUIContext**( )

> This function returns context provided by GUI configuration.
>
> > ***Returns***
> > *data* (string) - Data received using parsing table configured by GUI tool
> > *error* (integer) - Zero on success, positive value indicates line number from the request table defined by GUI, which caused the error
> > *acked* (integer) - According to GUI, data should be sent using acknowledged (1) or non-acknowledged (0) LoRa transport
> > *wake* (integer) - Number of minutes, according to GUI, the device should sleep using *wakeUpIn()* function.
> > *intArg* (integer) - Used with S0 callback onThreshold(), contains the number of S0 input, which raised the event.
>
> *example:*
>
> ```
> --get context provided by GUI configuration
> data,error,acked,wake = api.getGUIContext()
> ```

num = api.**getUniqueNumber**( )

> This function returns an unique integer number in range <0; 2^32>.
>
> > ***Returns***
> > *num* (integer) - Unique number
>
> *example:*
>
> ```
> --get an unique number
> num = api.getUniqueNumber()
> ```

value = api.**getVar**(*index*)

> This function returns persistent variable value, can be used between different wake up iterations.
>
> > ***Arguments***
> > *index* (integer) - Index of the variable to read, 0 to 15 is available for RAM variables (lost on reset) and 16 to 1039 is available for variables stored in EEPROM

> > ***Returns***
> > *value* (integer) - Value of the 32bit variable
>
> *example:*
>
> ```
> --get persistent variable value from index 1
> --can be used to send different data between wake-ups
> --for variables persistent between device reset,
> --use indexes 16 to 1039
> slotNumber = api.getVar(1)
> ```

api.**setVar**(*index*, *value*)

> This function saves a persistent variable value, can be used between different wake up iterations.
>
> > ***Arguments***
> > *index* (integer) - Index of the variable to read, 0 to 15 is available for RAM variables (lost on reset) and 16 to 1039 is available for variables stored in EEPROM
> > *value* (integer) - Value to store at *index*
>
> *example:*
>
> ```
> --set persistent variable value from index 3 to value of 3424
> --can be used to send different data between wake-ups
> --for variables persistent between device reset,
> --use indexes 16 to 1039
> api.setVar(3, 3424)
> ```

api.**setVerbosity**(*verbosity*)

> This function sets verbosity level.
>
> > ***Arguments***
> > *verbosity* (integer) - Zero removes all debug printing, verbosity increases up to value of four representing the maximum details
>
> *example:*
>
> ```
> --print only critical errors, other print outs are suppressed
> api.setVerbosity(1)
> ```

volt = api.**getBatteryVoltage**(*index*)

> This function provides a measured value of battery voltage in millivolts as return value.
>
> > ***Returns***
> > *volt* (integer) - Current battery voltage in millivolts

> *example:*
>
> ```
> --get battery voltage value in mV
> mv = api.getBatteryVoltage()
> ```

status = api.**wakeUpAt**(*day*, *hour*, *minute*, *second*)

> This function schedules the next wake up event of the device to provided day of month (*day*), *hour*, *minute* and *second*. The provided wake up date is therefore absolute and not relative as in *wakeUpIn()*.
>
> > ***Arguments***
> > *day* (integer) - Day of month, range 1 to 31
> > *hour* (integer) - Hour, range 0 to 23
> > *minute* (integer) - Minute, range 0 to 59
> > *second* (integer) - Second, range 0 to 59
> > ***Returns***
> > *status* (integer) - Execution status, 0 for success and -1 for error
>
> *example:*
>
> ```
> --schedules next wake up to the 25th, 2:22:58
> status = api.wakeUpAt(25, 2, 22, 58)
> ```

status = api.**wakeUpIn**(*day*, *hour*, *minute*, *second*)

> This function schedules the next wake up event of the device after specified time interval. The provided wake up date is therefore relative and not absolute as in *wakeUpAt()*.
> Note: The input arguments are not limited, but the total period specified must not exceed 31 days. (e.g. hour = 40, days = 2 gives a period of 3 days and 16 hours).
>
> > ***Arguments***
> > *day* (integer) - Day, range 0 to 31
> > *hour* (integer) - Hour, range 0 to X
> > *minute* (integer) - Minute, range 0 to X
> > *second* (integer) - Second, range 0 to X
> > ***Returns***
> > *status* (integer) - Execution status, 0 for success and -1 for error
>
> *example:*
>
> ```
> --schedules next wake up in 1 day and 122 minutes
> status = api.wakeUpIn(1, 0, 122, 0)
> ```

year, *month*,*day*,*hour*,*minute*,*second* = api.**getTimeDate**()

> This function returns current time running in the device. The time can be synchronized using LoRa or debug cable.

> > ***Returns***
> > *year* (integer) - Current year
> > *month* (integer) - Current month
> > *day* (integer) - Current day of month
> > *hour* (integer) - Current hour
> > *minute* (integer) - Current minute
> > *second* (integer) - Current second
>
> *example:*
> ```
> --read current date and time
> y,M,d,h,m,s = api.getTimeDate()
> ```

api.**dumpArray**(*str*)

> This function prints contents of variable as hexadecimal string.
>
> > ***Arguments***
> > *str* (string) - String with variable to be printed
>
> *example:*
> ```
> --print string "123ef" as hexadecimal
> api.dumpArray("123ef")
> 31 32 33 65 66
> ```

api.**S0setThreshold**(*channel*, *value*)

> This function defines a threshold between current value of a S0 channel counter and last reported value. When the difference of these last two reaches the *value*, the onThreshold() event is called.
>
> > ***Arguments***
> > *channel* (integer) - Number of the S0 channel, 0 to 3.
> > *value* (integer) - Threshold value, 0 disables the threshold, 0x1-0xFFFFFFFF sets the threshold.
>
> *example:*
> ```
> --sets threshold for channel 2 to the value of 10000
> api.S0setThreshold(2, 10000)
> ```

value = api.**S0readCounter**(*channel*)

> This function reports the current value of the S0 channel counter specified in the *channel* input argument.
> Note: By calling this function, an internal shadow variable for the channel counter is updated, so that the counter for onThreshold() event is reset.
>
> > ***Arguments***
> > *channel* (integer) - Number of the S0 channel, 0 to 3.

> > ***Returns***
> > *value* (integer) - Value of the S0 counter
>
> *example:*
> ```
> --reads the value of S0 channel 3 and stores to val variable
> val = api.S0readCounter(3)
> ```

Example Scripts

Default Script

To be used with the GUI based configuration of the device. A basic error handling is provided and the device wakes up as defined per GUI.

```
function onWake ()
    buf,err,ack,wake,intArg = api.getGUIContext()

    if err ~= 0 then
        print("Error occured on line" .. tostring(err))
        print("Sending error code to LORA")
        api.loraSend(ack,20000,tostring(err))
        print("Done sending")
    else
        print("Sending to LORA")
        api.loraSend(ack,20000,buf)
        print("Done sending")
        print("No error, sent to lora")
    end

    api.wakeUpIn(0,0,wake,0)
end
function onStartup()
    print("Starting up LUA interface...")
end
```

Script with time slots

This script sends two different MBus requests to two different devices at two different baud rates. The two devices share the same MBus. To be compliant with LoRa transmission duty cycle, each device is read out in its own time slot. Furthermore, the battery voltage information is sent every third time slot.

```
function onWake ()
    buf,err,ack,wake,intArg = api.getGUIContext()

    state = api.getVar(0)
```

```
    if state == 0 then
        api.mbusSetup(2400, 8, 2, 1)
        api.mbusState(1)
        status, ans = api.mbusTransaction(
         pack.pack('<b5', 0x10,0x12,0x34,0x56,0x16),
         4000, 1)
        api.mbusState(0)
        state = 1
    elseif state == 1 then
        api.mbusSetup(9600, 8, 2, 1)
        api.mbusState(1)
        status, ans = api.mbusTransaction(
         pack.pack('<b5', 0x10,0x78,0x9A,0xBC,0x16),
         3000, 2)
        api.mbusState(0)
        state = 2
    else
        volt = api.getBatteryVoltage
        ans = "Battery: " .. tostring(volt) .. "mV"
        wake = 2*wake --sleep twice more time
        state = 0
    end

    print("Sending to LORA")
    api.loraSend(ack,20000,ans)
    print("Done sending")
    api.wakeUpIn(0,0,wake,0)
    api.setVar(0,state)
end
function onStartup()
    print("Starting up LUA interface...")
end
```

Script for S0 inputs reporting

This script defines a LUA function wordToBuffer() for easy insertion of 32bit integer to a buffer and getS0Data() function, which is used to format a packet containing values from S0 counters and current battery voltage.
A new event is used - onThreshold(), which is called when an S0 channel is incremented by a defined amount of units (here the value is setup to 1000 in the onStartup() callback function). The data frame is sent to LoRa either periodically or when the threshold is hit.

```
function wordToBuffer(word)
    local buff = ""
    buff = buff ..  string.char(word%256) ..  string.char((word/256)%256) ..  string.cha
    return buff
end
```

```lua
-- get and format S0 inputs
function getS0Data()
    s00 = api.S0readCounter(0)
    print("S0-0: "..tostring(s00))
    s01 = api.S0readCounter(1)
    print("S0-1: "..tostring(s01))
    s02 = api.S0readCounter(2)
    print("S0-2: "..tostring(s02))
    s03 = api.S0readCounter(3)
    print("S0-3: "..tostring(s03))

    -- read old values
    s00_l = api.getVar(0)
    s01_l = api.getVar(1)
    s02_l = api.getVar(2)
    s03_l = api.getVar(3)

    s00_ll = api.getVar(4)
    s01_ll = api.getVar(5)
    s02_ll = api.getVar(6)
    s03_ll = api.getVar(7)

    -- update old values
    api.setVar(0, s00)
    api.setVar(1, s01)
    api.setVar(2, s02)
    api.setVar(3, s03)

    api.setVar(4, s00_l)
    api.setVar(5, s01_l)
    api.setVar(6, s02_l)
    api.setVar(7, s03_l)

    -- get battery voltage
    v = api.getBatteryVoltage()

    -- assemble the frame
    buf = string.char(5) -- device class
    buf = buf .. wordToBuffer(s00)
    buf = buf .. wordToBuffer(s00_l)
    buf = buf .. wordToBuffer(s00_ll)
    buf = buf .. wordToBuffer(s01)
    buf = buf .. wordToBuffer(s01_l)
    buf = buf .. wordToBuffer(s01_ll)
    buf = buf .. wordToBuffer(s02)
    buf = buf .. wordToBuffer(s02_l)
    buf = buf .. wordToBuffer(s02_ll)
    buf = buf .. wordToBuffer(s03)
    buf = buf .. wordToBuffer(s03_l)
    buf = buf .. wordToBuffer(s03_ll)
    buf = buf .. string.char(0)
    buf = buf .. string.char(v%256) ..  string.char((v/256)%256)
    buf = buf .. string.char(0)

    -- print the frame
    print("Frame in hex: <devClass, S0_0, S0_0_last, ... , 0, voltage, 0>")
    api.dumpArray(buf)

    return buf

end

function onWake ()
    buf,err,ack,wake = api.getGUIContext()
    print("onWake(), periodic wake up")
    buf = getS0Data()
        print("Sending to LORA")
        api.loraSend(ack,20000,buf)
        print("Done sending")
        print("No error, sent to lora")

    api.wakeUpIn(0,0,wake,0)

end

function onThreshold ()
        buf,err,ack,wake,src = api.getGUIContext()

        print("onThreshold(), reason S0: " .. tostring(src))

        buf = getS0Data()
        print("Sending to LORA")
        api.loraSend(ack,20000,buf)
        print("Done sending")
        print("No error, sent to lora")
end


function onStartup()
    print("onStartup(), Starting up LUA interface...")

    --set to threshold
    api.S0setThreshold(0, 1000)
    api.S0setThreshold(1, 1000)
    api.S0setThreshold(2, 1000)
    api.S0setThreshold(3, 1000)

    -- initialize old values
    api.setVar(0, 0)
    api.setVar(1, 0)
    api.setVar(2, 0)
    api.setVar(3, 0)
    api.setVar(4, 0)
    api.setVar(5, 0)
    api.setVar(6, 0)
    api.setVar(7, 0)
end
```

Other available LUA API

Math library

This library provides basic mathematic functions, note that for simplicity, lua in this embedded device uses integer and not float arithmetics.
Any function from this library is prepended by “math.”.
The list of supported functions is: **abs, ceil, floor, max, min, pow, random, randomseed, sqrt.**
For more details and function arguments definition, refer to official Lua 5.1 documentation:
https://www.lua.org/manual/5.1/manual.html (Section 5.6, mathematical functions)

Pack library

The pack library is used as a convenient way to parse binary buffers and to create a binary representation of lua variables.
Any function from this library is prepended by “pack.”.
The list of supported functions is: **pack, unpack.**
For more details and function arguments definition, refer to eLua project pages:
http://www.eluaproject.net/doc/v0.8/en_refman_gen_pack.html#overview

String library

This library can be used for manipulation of string variables and string buffers.
Any function from this library is prepended by “string.”.
The list of supported functions is: **byte, char, format, len, sub.**
For more details and function arguments definition, refer to official Lua 5.1 documentation:
https://www.lua.org/manual/5.1/manual.html (Section 5.4, string manipulation)

Lua base library

This library contains a basic lua 5.1 language library.
The list of supported functions is: **assert, collectgarbage, dofile, error, gcinfo, getfenv, getmetatable, loadfile, load, loadstring, next, pcall, print, rawequal, rawget, rawset, select, setfenv, setmetatable, tonumber, tostring, type, unpack, xpcall.**

For more details and function arguments definition, refer to official Lua 5.1 documentation:
https://www.lua.org/manual/5.1/manual.html (Section 5.1, basic functions)

Lua debug library

This library is used for debugging. Any function from this library is prepended by “debug.”.
List of supported functions is: **debug, getfenv, gethook, getinfo, getlocal, getregistry, getmetatable, getupvalue, setfenv, sethook, setlocal, setmetatable, setupvalue, traceback.**
For more details and function arguments definition, refer to official Lua 5.1 documentation:
https://www.lua.org/manual/5.1/manual.html (Section 5.9, the debug library)