Ixxat® BY HMS NETWORKS

ENGLISH

CAN@net NT/CANbridge NT Lua ADK

SOFTWARE DESIGN GUIDE

4.02.0332.20002
Version 1.5
Publication date 2023-11-27

HMS

Important User Information

Disclaimer

The information in this document is for informational purposes only. Please inform HMS Networks of any inaccuracies or omissions found in this document. HMS Networks disclaims any responsibility or liability for any errors that may appear in this document.

HMS Networks reserves the right to modify its products in line with its policy of continuous product development. The information in this document shall therefore not be construed as a commitment on the part of HMS Networks and is subject to change without notice. HMS Networks makes no commitment to update or keep current the information in this document.

The data, examples and illustrations found in this document are included for illustrative purposes and are only intended to help improve understanding of the functionality and handling of the product. In view of the wide range of possible applications of the product, and because of the many variables and requirements associated with any particular implementation, HMS Networks cannot assume responsibility or liability for actual use based on the data, examples or illustrations included in this document nor for any damages incurred during installation of the product. Those responsible for the use of the product must acquire sufficient knowledge in order to ensure that the product is used correctly in their specific application and that the application meets all performance and safety requirements including any applicable laws, regulations, codes and standards. Further, HMS Networks will under no circumstances assume liability or responsibility for any problems that may arise as a result from the use of undocumented features or functional side effects found outside the documented scope of the product. The effects caused by any direct or indirect use of such aspects of the product are undefined and may include e.g. compatibility issues and stability issues.

Contact Information

Postal address:
Box 4126
300 04 Halmstad, Sweden

E-Mail: info@hms.se

Table of Contents

This page is intentionally left blank.

1. Purpose of this Document

The Lua ADK FAQ answers common questions concerning the development of Lua applications for Ixxat CAN@net NT and CANbridge NT.

For information how to program in Lua, refer to Lua resources on the internet:

- complete language specification in the Lua Reference Manual on lua.org
- information about availability and licensing issues in the lua.org FAQ
- Q and A content and help for learning Lua as a second language in the unofficial Lua FAQ
- Lua Users Wiki provides examples source and relevant discussions, and information to start Learning Lua
- Programming in Lua by Roberto Ierusalimschy, one of the creators of Lua, is a detailed introduction to Lua programming. The online version is the first edition and aims at. Lua 5.0. The third edition for Lua 5.3 is available for purchase.

This FAQ is derived from the FAQ of the NodeMCU 3.0.0 framework, published under the license MIT© zeroday)/ nodemcu.com. This original FAQ was started by Terry Ellison as an unofficial FAQ in mid 2015. This version in October 2019 includes some significant rewrites.

2. Description of Ixxat Lua ADK

The Ixxat Lua ADK is based on the standard Lua 5.3.5 distribution, a fully featured implementation of Lua 5.3. The implementation is optimized for embedded system development and execution to provide a scripting framework that can be used to deliver useful applications within the limited RAM and Flash memory resources of embedded processors.

- All standard Lua language constructs and data types are working.
- Main Lua standard libraries `core` (partly), `coroutine`, `string`, `math`, `debug` and `table` are implemented.
- File handling functions like `require`, `dofile`, and `loadfile` are missing. To be able to load other Lua files, like json.lua and misc.lua, the `sys` library provides the `dofile` function that is similar to the original functions.

2.1. Difference between Standard Lua and Ixxat Lua ADK

Lua is primarily an extension language that makes no assumptions about a main program. Lua works embedded in a host application to provide a powerful, lightweight scripting language for use within applications. The host application can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code. Through the use of C functions, Lua can be augmented to cope with a wide range of different domains, thus creating customized programming languages that share a syntactical framework.

The Ixxat Lua ADK is a firmware extension and must be layered over the standard firmware. The hooks and features of Lua enable the Ixxat Lua ADK to be seamlessly integrated without losing any of the standard Lua language features. The firmware has replaced some standard Lua modules that do not align well with the firmware. For example, the standard `io` and `os` libraries do not work, but are partly replaced by the libraries `can`, `mqtt`, and `sys`. To write to the terminal window that is provided by the CAN-Gateway Configurator the function `print(string)` is used by default.

The main impacts of the Ixxat Lua ADK are how application programmers must approach the developing and structuring of their applications. The ADK is nonpreemptive and event driven. Tasks can be associated with given events by using the ADK API to registering callback functions to the corresponding events. Events are queued internally within the ADK, and then the associated tasks are called one at a time. Each task returns the control to the ADK when run to completion (finished successfully).

The ADK libraries act as C wrappers around registered Lua callback functions to enable these to be used as ADK tasks. Therefore an event-driven programming style must be used in writing Lua programs.

2.2. Lua Library Functions Enabling the Registration of Lua Callbacks

The following library functions define or use callbacks:

- `sys.call_after()` in Lua module `sys`
- `can.register_msg()` in Lua module `can`
- `mqtt.subscribe()` in Lua module `mqtt`

For `can` the standard Lua callback function `on_can` is used and for `mqtt` the standard Lua callback function `on_mqtt`.

The libraries (`can`, `sys`, `mqtt`, etc.) use the ADK callback mechanism to bind Lua processing to individual events (for example a CAN message reception). Developers should make full use of these events to keep Lua execution sequences short.

For more information see the module documentation in Lua API Function and Module Reference, p. 9.

2.3. Limitations of the Ixxat Lua ADK

The Ixxat Lua ADK has hardware resource limitations because code is loaded from the Flash file system into off-chip RAM and executed there. This RAM area is shared between Lua and other tasks on the target. The overall size is approximately 900 Kbyte. Therefore, Lua can use up to approximately **600 Kbyte** RAM for code and data.

The total amount of memory in use by Lua (in Kbytes) can be determined and written to the terminal window of the CAN-Gateway Configurator with the following command:

```
print(collectgarbage("count"))
```

Keep in mind:

- Lua scripts need much more CPU time than compiled code written in C.
- Do as less as possible computation in Lua.
- Do the computation with the lowest possible frequency.

To reduce the effort for Lua:

- If possible use mapping tables and other features.
- Use the `max_frequency` feature for registered can messages to reduce the `on_can` call frequency.
- Use the `array.repack` function to reassemble CAN message payload.

The needed CPU power can be monitored via the output in the status bar of the CAN-Gateway Configurator. The Lua function `sys.get_us_time()` allows to measure execution times more precisely.

A typical value for the execution time of a short Lua function is 150 to 200 µs. About 5000–8000 messages can be processed per second.

2.4. Specifics in Coding for the Ixxat Lua ADK

2.4.1. Bit Model 32 Bit Version

The Ixxat Lua ADK is compiled for the 32 bit model. Therefore integer and floating points values are stored as 32 bit values instead of 64 bit values as with a standard desktop Lua installation.

2.4.2. Errors and Debugging

The Ixxat CAN@net/CANbridge NT runs the Lua ADK over the native hardware. There is an underlying kernel system to capture errors and to provide graceful failure modes. To avoid affecting the standard firmware, error handling is kept simple. If an error occurs when calling a Lua function, an error message is written to the log file and Lua is stopped. Runtime errors in Lua functions are written to the terminal and Lua is stopped. The rest of the firmware is running without restrictions. To reactivate the Lua script the system has to be restarted.

There is currently no target debugging support. The print statement diagnostics through the terminal interface of the system must be used. The Ixxat Lua ADK provides a remote debugging feature to run the Lua script on the PC while communicating with the target device via an USB connection.

2.4.3. Declaring Variables

Two types of data can be assigned to Lua variables:

- values such as numbers, booleans, and strings
- references such as functions, tables, and user data

For example, if the contents of variable `a` are assigned to variable `b`, there is a difference between values and references. In case of a value, the content of `a` is copied into `b`. In case of a reference, both `a` and `b` now refer to the same object. No content is copied. The Ixxat Lua ADK provides the function `table.copy()` which returns a deep copy of the provided table parameters.

In standard Lua all variables can be classified as `globals` or `locals`. By default, any variable that is referenced and not previously declared as `local` is `global`. Global variables persist in the global table until the variable is explicitly deleted.

To see what global variables are in scope, use the following command:

```
for k,v in pairs(_G) do print(k,v) end
```

Local variables are lexically scoped. This means the scope of local variables is limited to the block where they are declared. A block is the body of a control structure, the body of a function or a chunk (the file or string with the code where the variable is declared). Any variable can be declared `local` within nested blocks or functions without affecting the enclosing scope. Because locals are lexically scoped it is also possible to refer to local variables in an outer scope. These variables are still accessible within the inner scope.

The Lua runtime internally uses hashed key access to retrieve keyed data from a table. Locals are stored as a contiguous vector and are accessed directly by an index, which is faster. Access to firmware based tables is particularly slow.

Using locals as follows is both a lot faster at runtime and generates less byte code instructions for their access:

```
local can_send = can.send
```

2.4.4. Passing Context between Lua Event Tasks

A single Lua function is bound to any event callback task. The function is executed from within the relevant C code using a `lua_call()`. Each function can invoke other functions and so on. But each function must ultimately return the control to the C library code which then returns the control to the firmware.

Lua local variables that are defined inside a function only exist within the context of this executing Lua function. Therefore locals are unreferenced on exit and any local data and garbage might be collected between the `lua_call()` actions.

Context can be passed in two ways between event tasks:

- by global variables: any global variable persists until it is explicitly dereferenced by assigning `nil` to the variable.
- by local variables: local variables that are defined in an outer scope (see description of locals in Declaring Variables, p. 3).

3. Using Lua ADK with the CAN NT Devices

The use of the ADK is only possible in Bridge operational modes.

The Lua ADK supports two operational modes:

- running the Lua script on the target device in autonomous mode (target mode)
- running the Lua script on the host PC for debugging purposes, communicating with the target device via USB (remote mode)

Functionally there is no difference between running the Lua script on the device or on the host PC. The operational mode can be set in the CAN-Gateway Configurator (see How to Enable the Use of the Lua ADK?, p. 5).

Running scripts on the target device provides only limited debugging possibilities via print statements. Running scripts on the host PC allows debugging with breakpoints and watch windows. Running Lua on the host PC provides the possibility to run huge Lua files with low computation time due to performance of PC, but the USB communication introduces extra delays for each interaction (e.g. CAN-Lua-CAN) of about one millisecond.

3.1. How to Enable the Use of the Lua ADK?

To be able to use the Lua ADK with the CAN NT device, the device must be updated with the Lua framework and Lua must be enabled in the CAN-Gateway Configurator.

1. Connect the device to the host computer and to power supply (for more information see User Manual of the device in use).
2. Make sure that the latest CAN-Gateway Configurator is installed (check within the product support pages on www.ixxat.com/support-bridges-gateways).
3. Start the CAN-Gateway Configurator and connect the device in use (for more information see User Manual CAN-Gateway Configurator).
4. In the toolbar open menu **Lua ADK** and select **Update Lua ADK**.
5. In **C:\Program Files\HMS\Ixxat CAN-Gateway Configurator V6** open the file lua_framework_vx and update the Lua ADK.
   - Lua ADK is loaded to the connected device.
6. In the configuration tree select **General** and enable the ADK in the drop-down list **Use of Lua as ADK**.
   a. enabled in target mode: running the script on the target device (see How to Use Lua in Target Mode?, p. 5)
   b. enabled in remote mode: running the script on the host PC, communicating with the device via USB (see How to Use Lua in Remote Mode?, p. 6)
7. In CAN-Ethernet-CAN bridges note that the Lua ADK must be enabled for the Master and each Slave individually.
8. In the toolbar open menu **Target** and select **Write configuration to target** to write the device configuration to the connected CAN NT device.

3.2. How to Use Lua in Target Mode?

1. In the CAN-Gateway Configurator enable the use of the Lua ADK in target mode (see How to Enable the Use of the Lua ADK?, p. 5).
2. In the toolbar open menu **Target** and select **Write configuration to target** to write the device configuration to the connected CAN NT device.

3. Create a Lua script.
   or
   Test the HelloWorld example, that is provided in **C:\Users\Public\Documents\HMS\Ixxat CAN-Gateway Configurator\Examples\Lua**.
4. Download the Lua script to the device with the CAN-Gateway Configurator or with the command line tool *CANGWfile.exe*.
   With CAN-Gateway Configurator
   a. In the toolbar open menu **Lua ADK** and select **Write Lua script to target**.
   b. It is possible to capture the print outputs of Lua with the terminal window.
      - Lua script is written to the CAN NT device.
      - Lua script is automatically started when the device is started.

   With command line tool CANGWfile.exe
   a. Make sure the CAN@net or CANbridge NT device is connected.
   b. To download and start the script in terminal mode use the following command:
      `CanGWfile.exe USB any w LUA myscript.lua -init -terminal`
      The command executes the following:
      - downloads the script myscript.lua to the connected CAN NT device
      - restarts the device and activates the script
      - flushes old text strings from the output FIFO on the target
      - opens the terminal mode for text outputs
   c. To terminate the tool press any key on the keyboard.
      - Lua script is automatically started when the device is started.

NOTE
For descriptions of the command line parameters see User Manual CAN-Gateway Configurator.

3.3. How to Use Lua in Remote Mode?

Running the Lua script on the host PC and using it in remote mode can be used for example for debugging. The recommend IDE is ZeroBrane Studio.

The Ixxat Lua ADK includes the command line tool LuaADK.exe. This is a Lua interpreter v5.3 including a USB driver for the communication with the CAN@net/CANbridge NT device. This tool has to be used instead of the Lua interpreter of the IDE.

To use the LuaADK.exe from ZeroBrane:

1. To open the user settings file, open menu **Edit — Preferences — Settings: user**.
2. Add the path to the LuaADK.exe (e.g. `path.lua53 = 'C:\\Program Files\\HMS\\IXXAT CAN-Gateway Configurator\\LuaADK.exe'`).
3. Open **Project — Lua Interpreter** and select **Lua 5.3**.
4. Restart ZeroBrane.

To communicate with the Target Device in Remote Mode

1. Make sure the CAN@net or CANbridge NT device is connected via USB.
2. Enable the use of the Lua ADK in remote mode (see How to Enable the Use of the Lua ADK?, p. 5).
3. In the CAN-Gateway Configurator, open menu **Target** and select **Write configuration to target** to write the device configuration to the connected CAN NT device.
4. Create a Lua script.

5. Note that the Lua script depends on the configuration on the target. For example, MQTT broker settings must be configured on the target to be able to use the callbacks in the Lua script.
6. Start the Lua script on the host PC.
   - LuaADK.exe reinitializes the device (connect device, test connection and load script).
7. Note that the target device only starts to operate, if the Lua script is started on the host PC.
8. Configure the target device from the Lua initialize function and start normal operation.
9. If ZeroBrane stops at function `initialize()`, use **Project — Continue** to run the Lua script or **Project — Step over** to single step the script line by line.

3.4. How to Debug the Lua File?

Lua is not able to detect set breakpoints, because functions like `loop`, `initialize`, and `on_can` are called from the C environment.

- To be able to detect set breakpoints, add the following line to each function:

```
function loop(ticks, elapsed)
    require('mobdebug').on()   -- <== Add this line at the beginning of
                                    the function
    -- my code
    ...
end
```

If this line is added, breakpoints can be set and single step debugging is possible.

NOTE
Remove the line when using the code in target mode.

4. Creating Lua Scripts

4.1. How does the Lua ADK Event System Work?

Understanding how the system executes the code can help you to structure the code better and to improve both performance and memory usage.

- Ixxat Lua ADK uses the tree kinds of callback functions `initialize`, `on_...`, and `loop`.
  - `initialize` is called once while the system is initializing. It is used to register further events and to set CAN message filters.
  - `loop` is called cyclically every 100 ms after the initialization phase, when the device is up and running. It is used to trigger cyclically recurring tasks.
  - `on_can` and `on_mqtt` functions are used to handle events from CAN and MQTT.
- Keep the function execution times as short as practical so that the overall system can work smoothly and responsively. The general recommendation is to keep the callbacks under 5 ms in duration. If exceeded intermittent problems because of mailbox overruns of the Lua task might occur.
- The Lua libraries provide a set of functions for declaring application functions (written in Lua) as callbacks to associate application tasks with specific hardware and timer events. These are also non-preemptive at an applications level. The Lua libraries work in consort with the ADK to queue pending events and invoke any registered Lua callback routines, which then run to completion uninterrupted. For example Lua`sys.call_after(time, callback, arg)` calls a function in the `sys` library which registers a Lua function for this timer alarm. When this alarm is triggered, it will call the Lua callback.
- Lua functions initiated by timer or network events and other callbacks run non-preemptively to completion before the next task can run. This includes all Lua ADK functions.

Observe that you are using the wrong approach,

- if you are not using timers and other callbacks.
- if you are using poll loops.
- if you are executing more than a few hundred lines of Lua per callback.

4.2. Example Applications

Various example applications can be found after installation of the CAN-Gateway Configurator download package in folder **C:\Users\Public\Documents\HMS\Ixxat CAN-Gateway Configurator\\Examples\Lua**.

Example application	Description
can_demo.lua	Shows how to handle (send and receive) CAN messages via Lua.
canfd_demo.lua	Shows how to handle (send and receive) CAN FD messages via Lua.
coroutine.lua	Implements a task and delay function and show how more than one task can be executed simultaneously.
hello_world.lua	Read and output ADK and device information, additionally blink with the user LED.
iso15765.lua	Segmentation of large CAN FD messages into classic CAN messages using the ISO 15765-2 transport protocol
mqtt.lua	Shows how to subscribe and publish MQTT messages.
mqtt_json.lua	Subscribes MQTT JSON messages and sends the messages on CAN and vice versa.
repack.lua	Repack the data bytes of a receive CAN messages from CAN1 and send on CAN2

5. Lua API Function and Module Reference

5.1. Lua Callback Functions Called from C

Function 'initialize'

```
initialize()
```

Called once while the firmware is being initialized. It is used to register CAN and MQTT messages for reception and to initialize variables.

Function 'loop'

```
loop(ticks, elapsed)
```

Called cyclically every 100 ms after the initialization phase, when the device is up and running. It is used to trigger cyclically recurring tasks.

- `ticks`: counter value that is incremented with each call of `loop`
- `elapsed`: time in ms since the last call of `loop`

Function 'on_can'

```
on_can(topic, port, format, ident, data)
```

Called on each CAN message reception, based on message registrations via `can.register_msg`.

- `topic`: user defined reference value used by `can.register_msg`
  `port`: CAN port number (`1` .. `4`), depending on the available CAN ports of the target device
- `format`: combination of three characters to specify the message format, e.g.:
  - `"csr"` means: "classic CAN", "standard identifier (11 bit)", "remote frame"
  - `"fed"` means: "CAN-FD", "extended identifier (29 bit)", "data frame"
- `ident`: CAN message identifier (`0` .. `0x7FF` for standard messages and `0` .. `0x1FFFFFFF` for extended messages)
- `data`: message data bytes provided as Lua table or as byte array (see module `array`)

Function 'on_mqtt'

```
on_mqtt(handle, topic, payload, qos)
```

Called on each MQTT message reception, based on message registrations via `mqtt.subscribe`.

- `handle`: user defined reference value used by `mqtt.subscribe`
- `topic`: message topic as string
- `payload`: message data as string
- `qos`: quality of service (`0`, `1`, `2`)

Function 'on_action'

```
on_action(param1, param2, param3)
```

The "CAN-Gateway Configurator" allows the definition of action rules with the action " Call Lua function 'on_action'". If this action is triggered, the function `on_action` is called.

- `param1`, `param2` and `param3` are strings or numeric values, depending on the definition in the action rule.

5.2. C Functions Called from Lua

5.2.1. Module misc

Functions from the module `misc` are loaded by default and are directly available.

Function 'hex'

```
hex(val)
```

Converts the parameter `val` into a hexadecimal representation.

Function 'dump'

```
dump(o, indent, nested, level)
```

Dumps values in a one-statement format.

For example, {test = {"Testing..."}} becomes:

```
{
    test = {
    "Testing..."
    }
}
```

The function also supports tables as keys, but not circular references.

- `indent`: specifies an indentation string, it defaults to a tab. Use the empty string to disable indentation.
- `nested`: internal only. Do not provide any value.
- `level`: internal only. Do not provide any value.

Function 'string.split'

```
string.split(str, delim, include_empty, max_splits, sep_is_pattern)
```

Splits the string `str` into a list of substrings, breaking the original string on occurrences of the given separator `delim` (character, character set, or pattern).

- `include_empty`: boolean, default: `false`, e.g. `"a,,,b":split(",")` returns `{"a","","",b"}`
- `max_splits`: number, if negative, splits are not limited, default: `-1`
- `sep_is_pattern`: boolean, specifies whether the separator is a plain string or a pattern (regex), default: `false`, e.g. `"a,b":split(",")` returns `{"a","b"}`

Function 'table.copy'

```
table.copy(tbl)
```

Returns a deep copy of `tbl`.

5.2.2. Module can

The module `can` is loaded by default and can is directly available.

Function 'can.dump_data'

```
can.dump_data(data)
```

Formats `data` as byte hex dump, like:

```
can.dump_data({1, 2, 3, 4})  --> [01, 02, 03, 04]
```

Function 'can.register_msg'

```
can.register_msg({
        handle = 1,
        port = 1,
        format = "std",
        ident = 0x100,
        data_as = "array",
        max_frequency = 500,
    })
```

Registers a CAN/CAN-FD message for reception via `on_can`.

- `handle`: user defined reference to the registered message, to be used inside `on_can`
- `port`: CAN port number (`1` .. 4), depending on the available CAN ports of the target device
- `format` is `std` for 11-bit, or `ext` for 29-bit identifiers
- `ident`: message identifier
- `data_as` possible values:
  - `"array"`: received message payload (data bytes) is provided as byte array (userdata)
  - `"table"`: received message payload (data bytes) is provided as Lua table
- `max_frequency`: value in msec for the "message load reduction filter" to reduce the CPU load. CAN messages is forwarded to Lua with the given maximum frequency. This parameter is optional. Default value is `0` (no filter is used).

Type `"array"` has better performance and is therefore the preferred type. Type `"table"` is more generic and provides more possibilities to work with the received data.

Function 'can.send'

```
can.send(port, format, ident, payload)
```

Sends a CAN message.

- `port`: CAN port number (`1` .. 4), depending on the available CAN ports of the target device.
- `format`: combination of three characters to specify the message format, e.g.:
  - `"csr"` means: "classic CAN", "standard identifier (11 bit)", "remote frame"
  - `"fed"` means: "CAN-FD", "extended identifier (29 bit)", "data frame"
- `ident`: message identifier
- `payload`: data bytes of the message as `array` or `table` (see "can. register_msg")

Function `'can.stop'`

```
can.stop(port)
```

Stop the CAN controller to prevent message reception/transmission on the port.

- `port`: CAN port number (`1` .. 4), depending on the available CAN ports of the target device.

Function `'can.start'`

```
can.start(port)
```

Start the CAN controller again.

- `port`: CAN port number (`1` .. 4), depending on the available CAN ports of the target device.

Function `'can.init'`

```
can.init(port, mode, baudA, baudD)
```

Initialize the CAN controller.

The CAN controller must be stopped beforehand and restarted after initialization.

Example:

```
can.stop(1)
can.init(1, "---", 250, 0)
can.start(1)
```

- `port`: CAN port number (`1` .. 4), depending on the available CAN ports of the target device.
- `mode`: combination of three characters to specify the CAN operational mode:
  - `"aci"` means: "active", "Classic CAN", "ISO" (`"---"` is also valid as an alternative)
  - `"lfn"` means: "listen-only", "CAN-FD", "non ISO" ("non ISO" only with CAN-FD)
- `baudA`: Classic CAN: Baud rate value in KBaud like 125, CAN FD: Baud rate value in KBaud for arbitration phase
- `baudD`: For CAN-FD: Baud rate in KBaud for data phsae. 0 for Classic CAN

Function `'can.set_crossbar_switch'`

```
can.set_crossbar_switch(rx_port, tx_port, turn_on)
```

Turns the device internal message stream on or off. Message streams are specified by the CAN ports (receive and transmit can port). The uni-directional message flow between these ports can be turned on and off with the crossbar switch.

- `rx_port`: CAN port where the messages are received (1..4)
- `tx_port`: CAN port where the messages shall be sent (1..4)
- `turn_on`: possible values: `true` or `false`

5.2.3. Module `mqtt`

The module `mqtt` is loaded by default and directly available.

Function `'mqtt.publish'`

```
mqtt.publish(topic, payload, qos)
```

Publishes (sends) an MQTT message to an MQTT broker. The broker settings have to be specified in the device configuration (MQTT Broker Settings).

- `topic`: message topic as string (maximum length is 256 characters)
- `payload`: message data as string or array (maximum length is 1536 characters)
- `qos`: quality of service (`0`, `1`, `2`)

TIP
If Lua is enabled in "remote mode", the maximum string length for topic and payload is 184 bytes each due to USB communication limitations.

Function `'mqtt.subscribe'`

```
mqtt.subscribe(handle, topic, qos)
```

Sends a SUBSCRIBE message to the MQTT broker to be able to receive messages on topics of interest.

- `handle`: user defined reference to the registered message, to be used inside `on_mqtt`
- `topic`: message topic as string (maximum length is 256 characters)
- `qos`: quality of service (`0`, `1`, `2`)

TIP
If Lua is enabled in "remote mode", the maximum string length for topic and payload is 184 bytes each due to USB communication limitations.

5.2.4. Module `sys`

The module `sys` is loaded by default and directly available.

Function `'device_info'`

Reads and provides device related information as Lua table. Here a code extract:

```
local info = device_info()
print(info.hw_version)      --> e.g.: "3.02.00"
print(info.fw_version)      --> e.g.: "6.00.00"
print(info.fpga_version)    --> e.g.: "1.01.00"
print(info.security_lvl)    --> e.g.: 1
print(info.config_name)     --> user provided configuration name
print(info.config_type)     --> e.g.: "Local Bridge"
print(info.device_type)     --> e.g.: "CAN@net NT 200"
print(info.serial_num)      --> e.g.: "HW906509"
print(info.device_name)     --> user provided device name
print(info.ip_address)      --> e.g.: "192.168.178.23"
```

Function `'sys.get_ms_time'`

```
sys.get_ms_time()
```

Returns the system time (32 bit value) in milliseconds.

Function `'sys.get_us_time'`

```
sys.get_us_time()
```

Returns the system time (32 bit value) in microseconds.

Function `'sys.call_after'`

```
sys.call_after(ticks, func, arg)
```

Calls the Lua function `func` after `ticks` system ticks (resolution is 1 ms). `arg` can be used as parameter for `func` and should be a numeric value.

IMPORTANT
There are maximum 16 timers available, which can be used in parallel.

Example:

```
function foo(val)
    print("delayed "..val)
end

sys.call_after(100, foo, 123)
```

Function `'sys.set_user_led'`

```
sys.set_user_led(pattern, ticks)
```

Sets the User LED of the device.

- `pattern` possible values:
  - `1` - turn off (set `ticks` to `0`)
  - `2` - set to red (set `ticks` to `0`)
  - `3` - set to green (set `ticks` to `0`)
  - `4` - set to orange (set `ticks` to `0`)
  - `5` - flash red for `ticks` ticks
  - `6` - flash green for `ticks` ticks
  - `7` - flash orange for `ticks` ticks
  - `8` - blink red (frequency in Hz is `50 / ticks`)
  - `9` - blink green (frequency in Hz is `50 / ticks`)
  - `10` - blink red/green (frequency in Hz is `50 / ticks`)
  - `11` - blink orange (frequency in Hz is `50 / ticks`)
  - `12` - blink red/orange (frequency in Hz is `50 / ticks`)
  - `13` - blink green/orange (frequency in Hz is `50 / ticks`)
- `ticks`: time in multiples of 10 milliseconds

Function `'sys.logging'`

```
sys.logging(severity, message)
```

Writes a test message into the LOG/ERR file, e. g. `sys.logging("E", "my log message")`.

- `severity`: possible values: e'X'eption, 'E'rror, 'W'arning or 'L'ogging
- `message`: text string

5.2.5. Module cyc

The `cyc` module allows stopping, updating and starting the cyclic transmission of CAN messages. CAN messages to be sent cyclically can be configured with the "CAN-Gateway Configuratior". These messages can then be controlled from Lua.

Function `'cyc.stop_msg'`

```
cyc.stop_msg(row)
```

Configured "cyclic messages" will be sent on secondary side as soon as the message on primary side is received once. To prevent the transmission of such a message, the function `cyc.stop_msg` can be used e. g. from within the Lua function `initialize`.

- `row`: The row number in the "CAN-Gateway Configurator" grid (0 to 127)

Function `'cyc.start_msg'`

```
cyc.start_msg(row)
```

To enable the transmission of cyclic message again, the function `cyc.start_msg` is used.

TIP
The transmission only begins when the first message has been received on primary side or the `cyc.update_msg` function has been called.

- `row`: The row number in the "CAN-Gateway Configurator" grid (0 to 127)

Function 'cyc.update_msg'

```
cyc.update_msg(row, format, ident, payload)
```

The function `cyc.update_msg` is used to start the transmission of messages and/or to update the message identifier or data for transmission.

- `row`: The row number in the "CAN-Gateway Configurator" grid (0 to 127)
- `format`: Combination of three characters to specify the message format, e. g.:
  - `"csr"` means: "classic CAN", "standard identifier (11 bit)", "remote frame"
  - `"fed"` means: "CAN-FD", "extended identifier (29 bit), "data frame"
- `ident`: message identifier
- `payload`: data bytes of the message as `array` or `table` (see Function 'can.register_msg', p. 11)

5.2.6. Module `json`

The module `json` is optional and has to be loaded via `json= sys.dofile('json.lua')`.

Function 'json.encode'

```
json.encode(val)
```

Serializes the provided value as JSON string. The result is returned. Allowed data types for `value` are `boolean`, `number`, `string`, and `table`.

Function 'json.decode'

```
json.decode(st)
```

De-serializes the provided JSON string. The result is returned.

5.2.7. Module `array`

The module `array` is loaded by default and directly available.

Provides the data type `array` which is a *userdata* byte array with up to 64 bytes, used for CAN messages payload.

The module provides functions to generate and modify byte arrays in C which is more efficient than doing this in Lua, like:

- `extract` data bytes from an array
- `insert` data bytes into an array
- `repack` data from one array into another

Function 'array.new'

```
a = array.new(64, 0)
-- OR
a = array.new({1, 2, 3, 4, 5 ,6 ,7 ,8})
```

Creates a new instance of type `array`. The first parameter is used as size (number of bytes) , the second parameter as initialization value for the array.

A provided table is used as list of initialization values.

NOTE
Array/table indexes in Lua always start with `1` (instead of `0` as in many other languages).

Function `'array.extract'`

```
a = array.extract(a, first [, last])
```

Returns an extract of `a` that starts at `first` and continues until `last`. If `last` is absent, then it is assumed to be equal to the array length.

Function `'array.insert'`

```
a = array.insert(a, [pos,] value)
```

Inserts element `value` at position `pos` in `a`, shifting up the elements `a[pos]` ···, `a[#a]`. The default value for `pos` is `#a+1`, so that a call `array.insert(a, x)` inserts `x` at the end of array `a`.

Function `'array.string_to_array'`

```
a = array.string_to_array("this is string")
```

Converts a Lua string to a byte array. The length of the array corresponds to the number of string characters.

Function `'array.array_to_string'`

```
s = array.array_to_string(a)
```

Converts a byte array to a Lua string. Strings in Lua are not zero-terminated and can contain any arbitrary binary data.

Function `'array.compile'`

```
format = array.compile(s)
```

Generates a binary format string for `array.repack()` based on the given ASCII format string.

`array.repack()` allows a re-packaging of data bytes from one array into another. The format string defines which bytes are used and how they are packed into the destination array.

Example:

```
-- copy 4 bytes to the destination array in opposite order
format = array.compile("#4,#3,#2,#1")
dest_array = array.repack(format, src_array)

-- this corresponds to
dest_array[1] = src_array[4]
dest_array[2] = src_array[3]
dest_array[3] = src_array[2]
dest_array[4] = src_array[1]
```

The format allows constant values in addition:

```
-- copy only 2 bytes from the source array
format = array.compile("#1,0,#2,0x55")
dest_array = array.repack(format, src_array)

-- this corresponds to
dest_array[1] = src_array[1]
dest_array[2] = 0
dest_array[3] = src_array[2]
dest_array[4] = 0x55
```

References to additional function parameters are also possible:

```
-- copy 2 byte variables and 2 bytes from the source array to the
   destination array
format = array.compile("@1,@2,#3,#4")
dest_array = array.repack(format, src_array, var1, var2)

-- this corresponds to
dest_array[1] = var1
dest_array[2] = var2
dest_array[3] = src_array[3]
dest_array[4] = src_array[4]
```

Important to know:

- The length of the destination array is directly given by the format string length.
- The maximum value for a reference (e.g. `#8`) is the source array length.
- The maximum number for variable references (e.g. `@1`) is 32.
- Valid value for variables are `0` to `255` (unsigned byte).
- The maximum value for a constant is `255`, or `0xff`.

Function `'array.repack'`

```
dest_array = array.repack(format, src_array)
```

Packs the data from the array `src_array` according to the compiled format string `format` and returns the result array.

Valid Operations on Arrays

Read a value:

```
val = a[2]
```

Write a value:

```
a[l] = val
```

Size of the array:

```
num_bytes = #a
```

Concatenate two arrays:

```
arr_c = arr_a .. arr_b
```

Iterate over an array:

```
for _,v in ipairs(a) do
-- do something
end
```

6. Lua License