RE866 Lua Software User Guide

1VV0301521 Rev. 0 – 2019-03-01

TELIT
TECHNICAL
DOCUMENTATION

SPECIFICATIONS ARE SUBJECT TO CHANGE WITHOUT NOTICE

NOTICE

While reasonable efforts have been made to assure the accuracy of this document, Telit assumes no liability resulting from any inaccuracies or omissions in this document, or from use of the information obtained herein. The information in this document has been carefully checked and is believed to be reliable. However, no responsibility is assumed for inaccuracies or omissions. Telit reserves the right to make changes to any products described herein and reserves the right to revise this document and to make changes from time to time in content hereof with no obligation to notify any person of revisions or changes. Telit does not assume any liability arising out of the application or use of any product, software, or circuit described herein; neither does it convey license under its patent rights or the rights of others.

It is possible that this publication may contain references to, or information about Telit products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that Telit intends to announce such Telit products, programming, or services in your country.

COPYRIGHTS

This instruction manual and the Telit products described in this instruction manual may be, include or describe copyrighted Telit material, such as computer programs stored in semiconductor memories or other media. Laws in the Italy and other countries preserve for Telit and its licensors certain exclusive rights for copyrighted material, including the exclusive right to copy, reproduce in any form, distribute and make derivative works of the copyrighted material. Accordingly, any copyrighted material of Telit and its licensors contained herein or in the Telit products described in this instruction manual may not be copied, reproduced, distributed, merged or modified in any manner without the express written permission of Telit. Furthermore, the purchase of Telit products shall not be deemed to grant either directly or by implication, estoppel, or otherwise, any license under the copyrights, patents or patent applications of Telit, as arises by operation of law in the sale of a product.

COMPUTER SOFTWARE COPYRIGHTS

The Telit and 3rd Party supplied Software (SW) products described in this instruction manual may include copyrighted Telit and other 3rd Party supplied computer programs stored in semiconductor memories or other media. Laws in the Italy and other countries preserve for Telit and other 3rd Party supplied SW certain exclusive rights for copyrighted computer programs, including the exclusive right to copy or reproduce in any form the copyrighted computer program. Accordingly, any copyrighted Telit or other 3rd Party supplied SW computer programs contained in the Telit products described in this instruction manual may not be copied (reverse engineered) or reproduced in any manner without the express written permission of Telit or the 3rd Party SW supplier. Furthermore, the purchase of Telit products shall not be deemed to grant either directly or by implication, estoppel, or otherwise, any license under the copyrights, patents or patent applications of Telit or other 3rd Party supplied SW, except for the normal non-exclusive, royalty free license to use that arises by operation of law in the sale of a product.

USAGE AND DISCLOSURE RESTRICTIONS

I. License Agreements

The software described in this document is the property of Telit and its licensors. It is furnished by express license agreement only and may be used only in accordance with the terms of such an agreement.

II. Copyrighted Materials

Software and documentation are copyrighted materials. Making unauthorized copies is prohibited by law. No part of the software or documentation may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, without prior written permission of Telit.

III. High Risk Materials

Components, units, or third-party products used in the product described herein are NOT fault-tolerant and are NOT designed, manufactured, or intended for use as on-line control equipment in the following hazardous environments requiring fail-safe controls: the operation of Nuclear Facilities, Aircraft Navigation or Aircraft Communication Systems, Air Traffic Control, Life Support, or Weapons Systems (High Risk Activities"). Telit and its supplier(s) specifically disclaim any expressed or implied warranty of fitness for such High Risk Activities.

IV. Trademarks

TELIT and the Stylized T Logo are registered in Trademark Office. All other product or service names are the property of their respective owners.

V. Third Party Rights

The software may include Third Party Right software. In this case you agree to comply with all terms and conditions imposed on you in respect of such separate software. In addition to Third Party Terms, the disclaimer of warranty and limitation of liability provisions in this License shall apply to the Third Party Right software.

TELIT HEREBY DISCLAIMS ANY AND ALL WARRANTIES EXPRESS OR IMPLIED FROM ANY THIRD PARTIES REGARDING ANY SEPARATE FILES, ANY THIRD PARTY MATERIALS INCLUDED IN THE SOFTWARE, ANY THIRD PARTY MATERIALS FROM WHICH THE SOFTWARE IS DERIVED (COLLECTIVELY “OTHER CODE”), AND THE USE OF ANY OR ALL THE OTHER CODE IN CONNECTION WITH THE SOFTWARE, INCLUDING (WITHOUT LIMITATION) ANY WARRANTIES OF SATISFACTORY QUALITY OR FITNESS FOR A PARTICULAR PURPOSE.

NO THIRD PARTY LICENSORS OF OTHER CODE SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND WHETHER MADE UNDER CONTRACT, TORT OR OTHER LEGAL THEORY, ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE OTHER CODE OR THE EXERCISE OF ANY RIGHTS GRANTED UNDER EITHER OR BOTH THIS LICENSE AND THE LEGAL TERMS APPLICABLE TO ANY SEPARATE FILES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THIRD PARTY LICENSES

The firmware of this module includes software components that are published under the following licenses:

Lua: http://www.lua.org/license.html
eLua: https://github.com/elua/elua/blob/master/LICENSE
FreeRTOS: http://www.freertos.org/license.txt
Micro-ecc: https://github.com/kmackay/micro-ecc/blob/master/LICENSE.txt
Lua-POSIX: https://github.com/luaposix/luaposix
fnmatch: https://sourceware.org/newlib/

This firmware binary includes a micro-ecc library with the following license:

Copyright (c) 2014, Kenneth MacKay
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;

LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This firmware binary includes the file name pattern matching function fnmatch() of the Newlib C library with the following license:

Copyright (c) 1989, 1993, 1994

The Regents of the University of California. All rights reserved.

This code is derived from software contributed to Berkeley by Guido van Rossum.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

APPLICABILITY TABLE

- RE866 LUA

CONTENTS

1. INTRODUCTION

1.1. Scope

This document describes the RE866 Lua module.

1.2. Contact and Support Information

For general contact, technical support services, technical questions and report documentation errors contact Telit Technical Support at:

- TS-EMEA@telit.com
- TS-AMERICAS@telit.com
- TS-APAC@telit.com

or

- TS-SRD@telit.com for global LoRa support

Alternatively, use:

http://www.telit.com/support

For detailed information about where you can buy the Telit modules or for recommendations on accessories and components visit:

http://www.telit.com

Our aim is to make this guide as helpful as possible. Keep us informed of your comments and suggestions for improvements.

Telit appreciates feedback from the users of our information.

1.3. Text Conventions

Danger – This information MUST be followed or catastrophic equipment failure or bodily injury may occur.

Caution or Warning – Alerts the user to important points about integrating the module, if these points are not followed, the module and end user equipment may fail or malfunction.

Tip or Information – Provides advice and suggestions that may be useful when integrating the module.

All dates are in ISO 8601 format, i.e. YYYY-MM-DD.

1.4. Related Documents

- [1] RE866 Lua API Documentation 30555ST10904A
- [2] RE866 Hardware User Guide, 1VV0301364 (EU), 1VV0301525 (NA)

2. SOFTWARE

2.1. Features

The solution provided combines some already established functionally of Telit Lua modules with an option to replace the “external” host of the module. This controls the module via UART by an “internal” host implementation that runs in a Lua script engine but uses the same AT command set - that is already established for modules controlled via UART.

Additionally, access to external sensors is provided via TWI and ADC access functions and some GPIO access is provided by the corresponding dedicated native Lua interfaces.

Following are the feature support:

Lua 5.3.5 support:

- Up to 32kB of flash and 20kB of RAM are provided to implement and run custom scripts
- Precompiled Lua binary code can be executed in place (XIP), right out of the internal flash memory to save system resources (no copy to RAM needed)
- Compiler on module, no cross compiler needed for development
- Scripts are automatically compiled at start time for ease of use
- Scripts can be precompiled on module and stored as binary code to save space and startup time
- Fully event driven operation possible, no idle loop needed
- Shell for organizing scripts in local file systems

LoRa support:

- Controlled via internal AT command interface
- Configurable network configuration parameters

HW access:

- Fully automatic power management
- Supports 6 configurable digital I/O pins
- Supports 12-bit ADC analog input
- Supports TWI bus master with 100/400 kHz clock

Three on-board file Systems:

- 28kB flash file system for main application code and data
- 4kB flash file system for personalization/calibration data and special application code
- Volatile RAM file system for temporary files, e.g. when compiling scripts
- Supports multiple scripts and libraries in system
- Supports script auto start at system startup with priority management

Floating point operations:

- 32-bit floating point
- Cortex M4F HW supported operations
- Standard Lua math operations support out of the box

2.2. Software Block diagram

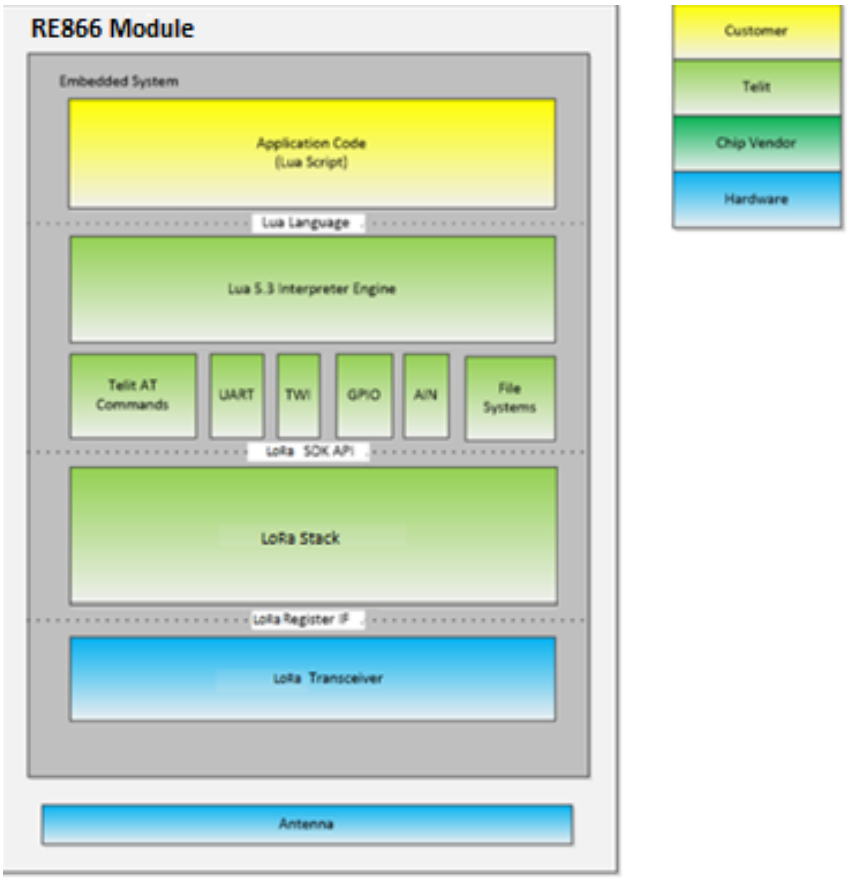

Figure 1 LoRa Software Block Diagram

3. SOFTWARE USAGE

3.1. Lua Language References

For a Lua introduction and command description the following online resources are recommended:

Official Lua website: https://www.lua.org

Online Lua command description: https://www.lua.org/manual/5.3/

As a printed reference and tutorial medium, the book “Programming in Lua, fourth edition” from Roberto Ierusalimschy (ISBN: 9788590379867) is recommended.

For other printed literature, refer: https://www.lua.org/pil/

3.2. PC Software Tools

Developing application code based on Lua scripts does not need a sophisticated toolchain but can be done with a simple tool setup with any text editor to write the Lua scripts and a basic terminal program for communication with the module via UART.

Telit provides a full IDE environment for Lua script development with LoRa. It can be downloaded from Telit Website.

For an ease of access, start and first evaluation an open source software can be used. Specific requirements for the tools are quite relaxed:

- The editor must be able to save ASCII Text files (e.g. *.TXT files)
- The terminal program should be able to upload files via XMODEM protocol

There are several other tools available, like:

1. “Notepad++” Editor - a basic, fast and easy to use text editor that supports Lua syntax highlighting: https://notepad-plus-plus.org
2. “TeraTerm” Terminal Program - A basic terminal program that allows to communicate via COM Port with the module and it supports XMODEM for script upload.

You may download the TeraTerm terminal program from the official open source web site:

https://en.osdn.jp/projects/ttssh2/releases/

To open the COM port to the module with Tera Term:

1. Click on “File ➔ New connection” in Tera Term main menu
2. Select “Serial” and the COM port the system created for the module plugged in
3. Click on “Setup ➔ Serial Port…” in Tera Term main menu
4. Configure to 115.200Baud /8Bit/No parity/1StopBit/Hardware flow control
5. If the module is connected and you press the [Enter] key on your keyboard the module should respond with a “#” prompt.

To upload a Lua script to the module with Tera Term:

1. Enter the shell command “recv” and the full path (including the file name), where to store the script on the module (e.g. “`recv /wo0/hello_world.lua`”, see chapter 3.4.2)
2. Click on “File ➔ Transfer ➔ XMODEM ➔ send” in Tera Term main menu

3. Select the Lua script you want to upload to the module
4. If your script is uploaded correctly you can execute it with the shell command “lua” on the module (e.g. “`lua /wo0/hello_world.lua`”, see chapter 3.4.3)

Other Lua Tools:

In general, there is a variety of tool and toolchain available which is mostly open source or freeware. Following link gives a sample list of such tools and tool chain:

http://lua-users.org/wiki/LuaEditorSupport

3.3. Software Startup

The module can be configured to start in four different modes based on the configuration of the Boot0 and #Testmode signals, namely:

1. Normal FW startup,
2. Boot Mode,
3. DTM Mode,
4. Test Mode.

Refer to [2] for more information.

For normal Firmware startup, the firmware will open the UART with 115.200kBaud/ 8 data bits/no parity/1 stop bit (115.200\8\N\1) and check for script auto execution definitions (see section 3.4.4 of this document). If no script is configured for auto execution or auto execution is cancelled by a Host, the eLua shell will be started automatically and signals '#' prompt.

3.4. Shell for File System Management

3.4.1. Shell Command List and Description

The shell supports a set of commands to control the module and its file systems.

Following are the list of commands with basic description. Additional information for each of these commands can be fetched by using the "help" command of the shell.

Example:

```
# help luac
luac - compile Lua script to Lua binary
Usage: luac [options] [filenames]
Available options are:
  -l       list (use -l -l for full listing)
  -o name  output to file 'name' (default is "/ram0/luac.out")
  -p       parse only
  -s       strip debug information
  -v       show version information
  --       stop handling options
  -        stop handling options and process stdin
#
```

The shell supports the following commands:

help	Shows a list of all shell commands or a description for the command given as an argument to ‘help’
lua	Start a Lua script. Without argument, an interactive Lua session is started. The prompt changes to “>”. Press CTRL+D or enter “os.exit()” to return to the shell
luac	Compiles a Lua script to a Lua binary
cat	List the contents of a file or multiple files
cp	Copy files
df	Shows free space on file systems
dir	Lists files and directories
echo	Display a line of text
ls	Lists files and directories
mv	Move/rename files
recv	Receive files via XMODEM or in raw binary
rm	Remove files
type	Lists the contents of a file or multiple files
mount	Mounts a file system
umount	Unmounts a file system
crc16	Displays a CRC16 checksum for a file
fdinfo	Shows information about open files
heapinfo	Shows information about free heap memory
stackinfo	Shows information about stack memory usage
ver	Show version information
set	Sets shell options
wofmt	Formats a write-once file system

3.4.2. Script File Upload

For uploading a Lua script (or any other type of file) to the module, the shell command “recv” can be used.

A file upload can be done via XMODEM protocol which is supported by most terminal programs (see chapter 3.1 for recommendations) that usually allow the user to comfortably select a file on the PC for XMODEM upload. The terminal program may send 128 bytes or 1 kbyte frames. Both are supported by the module. If the terminal program doesn’t support the XMODEM-CRC option, append the parameter “-P xmodem” to the “recv” command to disable CRC.

Alternatively, the upload can be done in raw binary mode - append the parameter “-P binary” plus the exact length in bytes of the file. Parameter of “recv” the destination directory and filename must be specified.

Once started, recv command will prompt “Waiting for file ...” and then one “C” character every second to indicate that it is ready to receive a file.

After the completion of the upload, the uploaded files can be listed by the “dir” or “ls” shell command.

Example:

```
# recv /wo0/hello_world.lua
Waiting for file ... CCC
done, got 19 bytes
received and saved as /wo0/hello_world.lua
#
```

The script can be uploaded to a nonvolatile flash file system (“/wo0” or “/wo1” => write once) or a volatile RAM based file system (“/ram0”).

Note: The RAM based file system dynamically allocates memory from the same heap memory as Lua, therefore reducing the amount of free memory that is available to Lua scripts. The main purpose of the RAM based file system is to allow the upload of Lua script in human readable format (*.lua) and to use the “luac” shell command to precompile and store it in the nonvolatile file system as precompiled Lua binary file (*.lc).

Example:

```
# luac -s -o /wo0/hello_world.lc /ram0/hello_world.lua
# rm /ram0/hello_world.lua
```

This example would compile the human readable “hello_world.lua” script located in the “/ram0” file system and store it as “hello_world.lc” Lua binary in the nonvolatile “/wo0” file system.

It would then delete the “hello_world.lua” script to free the dynamically allocated RAM occupied by using the shell command “rm”.

Note: For the file systems (“/wo0” or “/wo1”) the shell command “wofmt” must be used to remove all files in that file system and free the storage space occupied by the files. The “rm” command would just invalidate the file information but not free the allocated flash memory.

3.4.3. Script Execution

To execute a script that is located on the module, the shell command “lua” is used. Parameter “lua”, the path of the script must be specified to start.

Example:

```
# lua /wo0/hello_world.lua
Hello world
#
```

“lua” command can be used to run human readable Lua scripts as well as precompiled Lua bytecode.

If “*.lua” script is started, the script becomes compiled to RAM first. This causes a delay in execution of the code and leads to the lower availability of the RAM for application data runtime.

If Lua bytecode is started, the precompiled code is executed in (XIP) place in the flash memory of the module. This reduces the delay and more RAM is available. However, it might run slightly slower than the code in RAM due to flash memory wait states.

To generate and store a precompiled script file, refer command “luac”

3.4.4. Startup Behavior and Application AutoStart Setup

The startup sequence of the module firmware is customized by using shell script. If either one of the files (/wo0/init.sh and /wo1/init.sh) does not exists in the startup, then the firmware runs the default script /boot/init.sh.

On a module with a blank file system, this default script just enables the UART with 115.200 kbaud, prints the firmware version string and starts an interactive shell.

A default startup script allows to define a shell command line that is automatically executed instead of starting an interactive shell. It looks for a file that is named “startup.cmdline” in /wo0 and /wo1 (in this order). In case such a file is found, its first line will be executed by the shell without further host or user interaction.

For example, when the file “/wo0/startup.cmdline” exists and has the content “lua /wo0/autorun.lc”, then the precompiled binary /wo0/autorun.lc will be started automatically after a reset or power cycle of the module.

It is a good practice to implement a custom specific mechanism (e.g. waiting for a specific character on the UART) to terminate an execution of such an autorun script comfortably at least during development.

If such custom mechanism is not present, the execution of the command line in the “startup.cmdline” file can be suppressed by constantly sending “7” characters (Hex 0x37) while module startup after reset or power cycle. If the “7” characters are recognized after startup, the shell will start normally in interactive mode and prompt “#”.

When the automatically executed command line (from the “startup.cmdline” file) terminates without error, the shell takes over in interactive mode and prompts “#”.

When the automatically executed command line terminates with an error code (e.g. if Lua crashed with an out-of-memory exception), the module checks if a file named “recover.cmdline” exists. It searches the file systems for that file in this order: /ram0, /wo0, /wo1. If such a file is found, its first line will be executed by the shell. This mechanism can be used to restart your script in case of an unexpected error or to start another script to

recover from this situation. If the command line from the “recover.cmdline” file terminates with an error code, it is started again and again until it terminates without error. Although on each iteration the “recover.cmdline” file is searched for and read anew. This means, the script can change the file. If no “recover.cmdline” file is found, the shell takes over in interactive mode.

The behavior described above is implemented in the /boot/init.sh (the default startup shell script). After the development process, it might be required to prevent the end-user to gain access to the interactive shell, so the Lua scripts on the module can’t be read out. This is done by providing a custom init.sh script, either in /wo0 or /wo1, which simply does not start an interactive shell. The easiest way to do this is to copy the contents of the default script /boot/init.sh and remove the last line: “exec -“. This is the command that starts the interactive shell. If such a modified script exists in the local file system, the module is basically locked after the next reset. To gain access to the module later, both file systems (/wo0 and /wo1) must be fully erased (and with them the custom init.sh). This can be accomplished using the “erasefs” command when booting into the test mode. Please refer to document [2] on how to enter test mode.

4. LUA FOR APPLICATION DEVELOPMENT

4.1. Using Lua

The module implements the Lua version 5.3.5 where all the resources and definitions are already included in the module software, no additional libraries, header files or other information is needed.

A Lua script that prints a “Hello world” string is just one line:

```
print "Hello world"
```

For a Lua introduction and command description the following online resources are recommended:

Official Lua website: https://www.lua.org

Online Lua command description: https://www.lua.org/manual/5.3/

As a printed reference and tutorial medium the book “Programming in Lua, fourth edition” from Roberto Ierusalimschy (ISBN: 9788590379867) is recommended.

The module with Lua implementation provides control of the following resources:

Example:

```
local my_file = io.open("/wo0/mydata.txt", "w") -- write as text
my_file:write("123\n") -- writes "123\r\n"
my_file:close()
my_file = io.open("/wo0/mydata.txt", "r") -- read as text
local str = my_file:read("*l") -- returns "123", drops "\r\n"
print(#str) -- prints 3
my_file:close()
my_file = io.open("/wo0/mydata.txt", "rb") -- read as binary
str = my_file:read("*l") -- returns "123\r", drops "\n"
print(#str) -- prints 4 (counts the <CR>)
my_file:close()
```

4.1.1. Accessing UART interface

The UART is primarily used during Lua script development. On a blank module, a Unix command line interface, a shell, provides access to the file systems, manual script execution, or even allows the developer to start an interactive Lua command interpreter. Lua scripts running on the module may also use the UART for printing debug messages during development. Calling the `print()` function is all that is needed, since the standard I/O streams are redirected to the UART- if the module is accessed through the shell. Later, when the development is completed, it is required to disable the UART due to its power consumption. If the script still needs access to the UART, it may open the UART device file /dev/ttyS0 directly and configure it as shown in the following example:

Example:

```
local posix = require "posix"
local com = io.open("/dev/ttyS0", "r+b")
local com_cfg = { cflag = posix.termio.CRTSCTS,
```

```
                    lflag = 0,
                    iflag = posix.termio.ICRNL,
                    oflag = posix.termio.OPOST | posix.termio.ONLCR,
                    ospeed = posix.termio.B115200 }
local com_fd = posix.stdio.fileno(com)
posix.termio.tcsetattr(com_fd, posix.termio.TCSAFLUSH, com_cfg)
com:write "Hello world\n"
com:close()
```

For further information, please refer to [1] and for script examples, please refer to “Lua sample script package” that is part of the SDK.

4.1.2. Accessing GPIO Interface

To access GPIO functionality the Lua package “dio” is included in the firmware.

To use this package a `xyz=require "dio"` statement is needed in the script, where [xyz] defines the identifier to address GPIO functionality:

Example:

```
local dio = require "dio"
local pin_config = {[1] = { DIR = "in", PULL = "pd" }}
dio.port1:config(pin_config)
-- (...)
print("DIO1: " .. dio.port1:get(1))
```

For further information, please refer to [1] and for script examples, please refer to “Lua sample script package” that is part of the SDK.

4.1.3. Accessing TWI Interface

To access GPIO functionality the Lua package “twi” is included in the firmware.

To use this package a `xyz=require "twi"` statement is required in the script, where [xyz] defines the identifier to address TWI functionality.

Example:

```
local twi = require "twi"
twi.bus1:config("master", 100000) -- use 100kHz clock
twi.bus1:transfer(0x10, "abc", 0)
twi.bus1:config("")               -- disable TWI again
```

For further information, please refer to [1] and for script examples, please refer to “Lua sample script package” that is part of the SDK.

4.1.4. Accessing ADC Interface

To access ADC functionality, the Lua package “adc” is included in the Firmware.

To use this package a `xyz=require "adc"` statement is needed in the script, where [xyz] defines the identifier to address ADC functionality.

Example:

```
local adc = require "adc"
local conf = { [1] = { SRC = 1, REF = "int", GAIN = 1/5 } }
adc.adc1:config(conf)
print("Analog in: " .. adc.adc1:get(1))
```

For further information, please refer to [1] and for script examples, please refer to “Lua sample script package” that is part of the SDK.

4.1.5. Accessing LoRa AT command interface

For further Information, please refer to [1] chapter 3.5.

4.1.6. Reading and writing files

The standard Lua file I/O functions are used to access the files in local file systems.

Example:

```
local my_file = io.open("/wo0/mydata.txt", "w")
my_file:write("Hello world\n")
my_file:close()
my_file = io.open("/wo0/mydata.txt", "r")
print(my_file:read("*l"))
my_file:close()
```

For compatibility with files copied from Windows systems, by default, files are treated as text files by the file I/O layer. This means, the module software represents newlines (`"\n"`) as <CR><LF> (0x0d 0x0a) when written to files and interprets <CR><LF> as newlines when read from files. Yet single <LF> characters are also accepted as newlines when read. The actual content of the file /wo0/mydata.txt in the example above would be in hexadecimal:

48 65 6c 6c 6f 20 77 6f 72 6c 64 0d 0a

Mapping of newline characters would be problematic when handling raw binary data. For this reason, the letter “b” (for binary) may be appended to the mode string when opening a file. This disables the newline character mapping for read and write operations issued on the return file object.

4.2. Script Implementation

4.2.1. Memory Usage and Segmentation

Lua scripts are human readable text files. To execute a script, it is translated (compiled) to a binary representation first. If a Lua script stored on the module is started, the firmware will automatically compile the script and place the resulting bytecode in the RAM of the module for execution. RAM resources are very limited in the module. The amount of RAM occupied by the bytecode will not be available for application data at runtime.

To prevent the bytecode from taking up too much RAM, Lua scripts can be precompiled and stored in one of the nonvolatile write-once file systems (/wo0, /wo1). Precompiled code stored in these file systems can be executed in place (XIP) without getting moved to RAM first. See chapter 3.4.2 to learn about how to compile a script to bytecode.

Lua is a highly dynamic scripting language that automatically handles the allocation and deallocation of RAM resources. Due to this, fragmentation of RAM is unavoidable. This should be kept in mind when developing Lua scripts, because of the limited availability of RAM. To avoid resource shortage at runtime, it is a good practice to follow general rules for programming to avoid fragmentation.

For example, define variables/objects at the beginning of the code and initialize them with default values to allocate the resources needed at startup time.

4.2.2. Performance and Realtime Behavior

The RE866 is a Cortex-M4F driven embedded system that runs a LoRa stack and radio which has a high priority real-time requirement, a flash-based file system that might block the system during write operations and a FreeRTOS based Lua engine with a single CPU core. Due to this, Lua scripts with real-time requirements must be implemented carefully by considering these facts.

Just like any other programming language, Lua allows one and the same functionality being implemented in many ways. Although the functionality of different implementations might be the same, the efficiency and performance can be similar.

For clarification, a simple use case using different approaches to compare the resulting performance has been implemented - to toggle one digital I/O pin as fast as possible in a loop and monitor the frequency of the signal generated.

Following are the results of some demonstrations:

Note: The values may differ due to later optimization of the firmware

Reference implementation:

```
dio = require "dio"
while true do
      dio.port1:set(2, true)
      dio.port1:set(2, false)
end
```

+15% frequency increase compared to reference implementation:

```
dio = require "dio"
while true do
```

```
      dio.set(dio.port1, 2, true)
      dio.set(dio.port1, 2, false)
end
```

+30% frequency increase compared to reference implementation:

```
dio = require "dio"
local myport = dio.port1
while true do
      myport:set(2, true)
      myport:set(2, false)
end
```

+70% frequency increase compared to reference implementation:

```
dio = require "dio"
local myset = dio.set
local myport = dio.port1
local x
while true do
      x = not x
      myset(myport, 2, x)
end
```

+80% frequency increase compared to reference implementation:

```
dio = require "dio"
local myset = dio.set
local myport = dio.port1
while true do
      myset(myport, 2, true)
      myset(myport, 2, false)
end
```

5. DEVELOPMENT WITH APPZONE SDK

AppZone SDK version 4.0.5 provides an easy interface to write Lua5.3.5 scripts for RE866 module.

AppZone Lua environment extends the product list of Telit New Project Wizard with RE866 as an additional choice. Following is the screenshot:

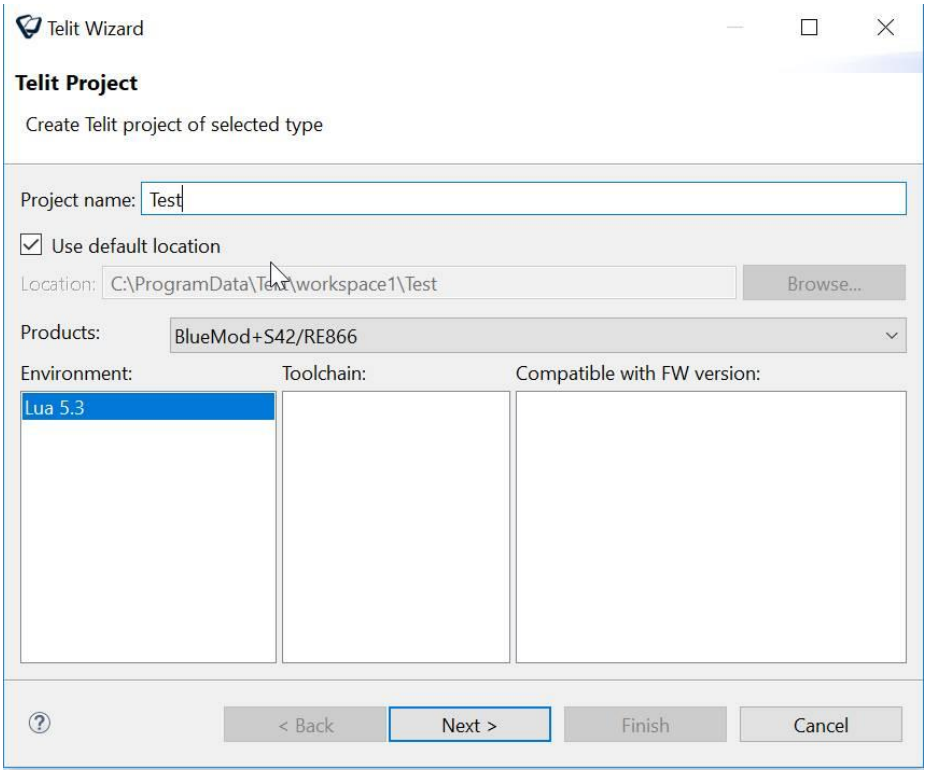

Figure 2:- Selection of RE866 module with Lua5.3

The wizard comes with a set of Lua scripts compatible with RE866 application development interface that users can either use as examples or as a skeleton of their applications.

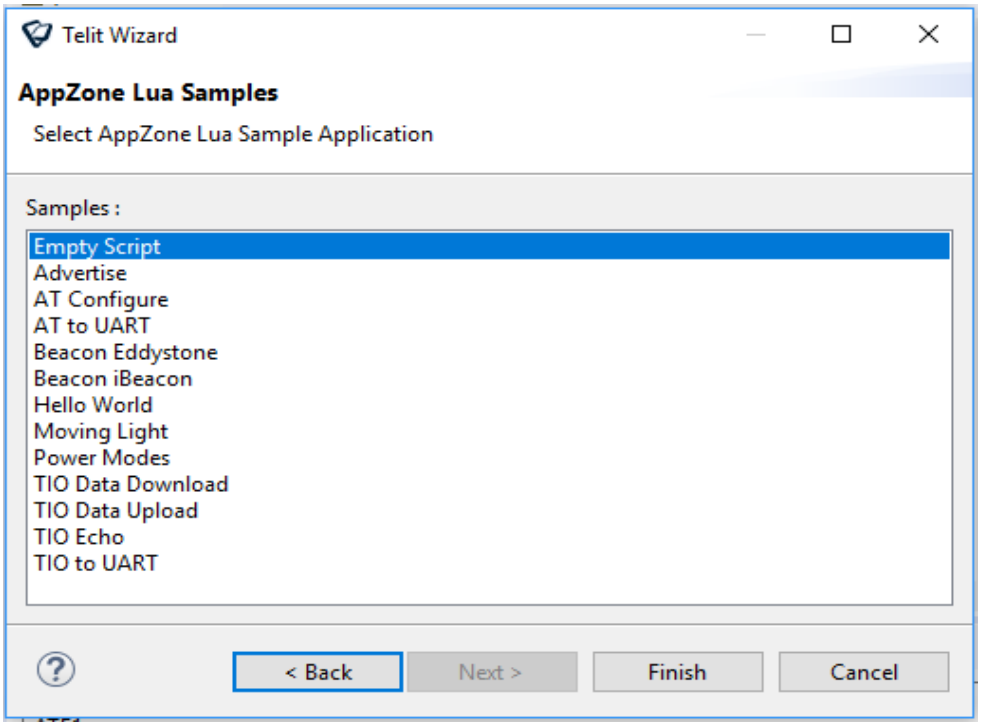

Figure 3:- Selection of Development Application Interface

Users can either test their scripts locally or deploy them on RE866 module using the Telit AT Console for Lua. Testing locally can be done by using the “Build and Run” standard Eclipse button, whereas deploying the script on the module can be done by dragging and dropping it to Lua console given that RE866 module is already connected.

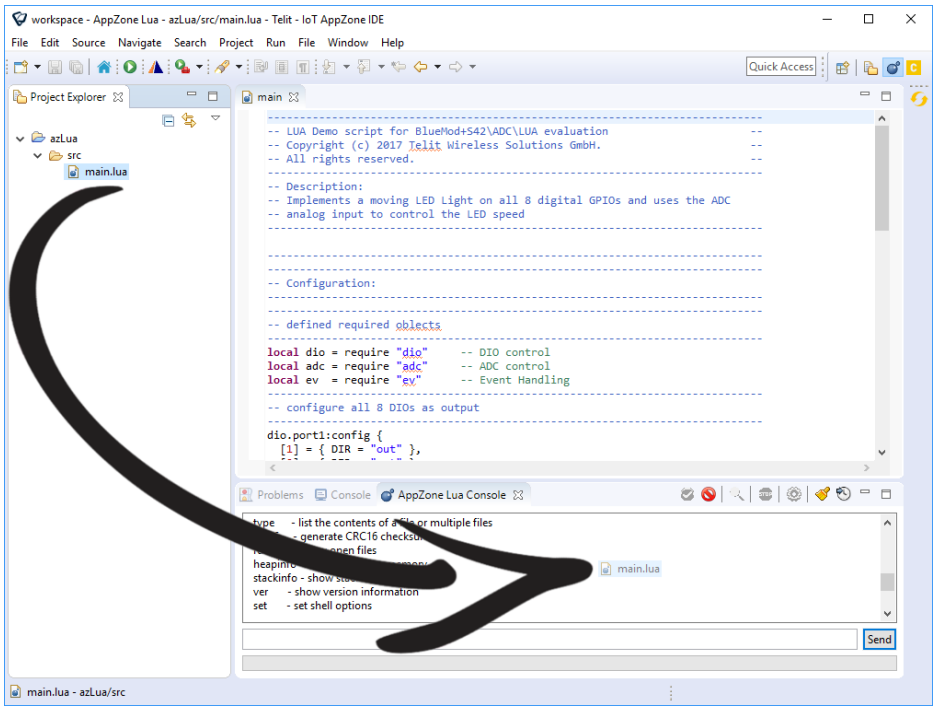

Figure 4:- Telit AT Console for LUA

RE866 module has two WO file systems (wo0 and wo1) and users can select the WO file system to deploy the application through the settings dialog of Lua AT console.

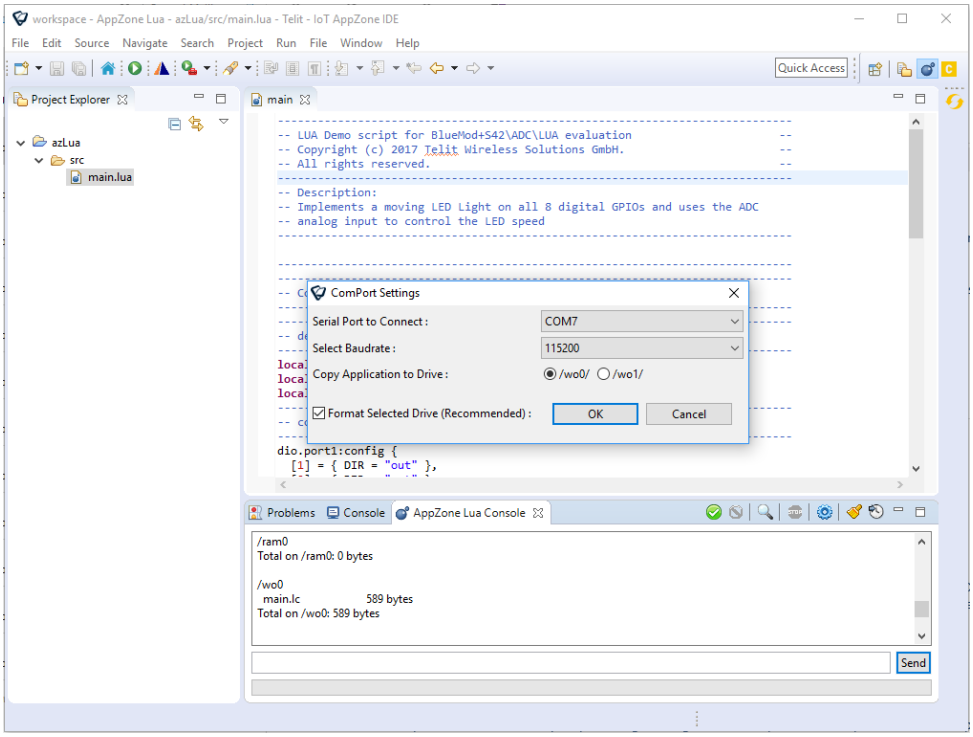

Figure 5:- Deploying the Application

When the script is dropped into AZ console, the SDK does the following actions:

1. Format selected wo/ drive
2. Copy application to RAM
3. Compile application on RAM
4. Copy application to formatted wo/ drive
5. Clean RAM
6. Run the application on board

6. GLOSSARY AND ACRONYMS

ADC	Analog to Digital Conversion
LoRa	Long Range
Lua	(pronounced LOO-ah) means "Moon" in Portuguese
TWI	Two wire serial interfaces
AZ	App Zone

7. DOCUMENT HISTORY

Revision	Date	Changes
0	2019-02-13	First issue.

SUPPORT INQUIRIES

Link to **www.telit.com** and contact our technical support team for any questions related to technical issues.

www.telit.com

Telit Communications S.p.A.
Via Stazione di Prosecco, 5/B
I-34010 Sgonico (Trieste), Italy

Telit IoT Platforms LLC
5300 Broken Sound Blvd, Suite 150
Boca Raton, FL 33487, USA

Telit Wireless Solutions Inc.
3131 RDU Center Drive, Suite 135
Morrisville, NC 27560, USA

Telit Wireless Solutions Co., Ltd.
8th Fl., Shinyoung Securities Bld.
6, Gukjegeumyung-ro8-gil, Yeongdeungpo-gu
Seoul, 150-884, Korea

Telit Wireless Solutions Ltd.
10 Habarzel St.
Tel Aviv 69710, Israel

Telit Wireless Solutions
Technologia e Servicos Ltda
Avenida Paulista, 1776, Room 10.C
01310-921 São Paulo, Brazil