Lua modules

Guide version: 1.0

The AMS111 IV Linux uses Lua 5.3.5 as scripting language

“Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.”

Lua Reference manual: https://www.lua.org/manual/5.3/

1. Modules organization

The Lua modules are automatically loaded from the following file:

`/home/user/picolo4modules/<module directory>/main.lua`

The data logger automatically calls the function `initModule()` from main.lua file. This call is performed at data logger startup. It is possible to add other Lua files to the same directory, but the main.lua must include them with command `require`.

Example:

If there are two files:

```
/home/user/picolo4modules/testModule1/main.lua
/home/user/picolo4modules/testModule1/utils.lua
```

Then the data logger starts only main.lua and it must include the following code in order to load utils.lua file:

```
require("utils")
```

The initModule function can contain initialization code. It can setup event handlers and initialize variables.

2. Built-in functions

It is possible to call functions of data logger software from Lua scripts. These functions are prefixed with mstep.

Function mstep.ReadChar

Reads from the data logger a variable of 1-byte character type.

Usage:

```
value, status = mstep.ReadChar("var")
```

`value` contains the value of the variable, or `nil`, if no data is available or the type is not correct.
`status` contains the variable status message in case of error.
`"var"` is the variable name in data logger.

Function mstep.ReadInt

Reads from the data logger a variable of 16-bit integer type.

Usage:

```
value, status = mstep.ReadInt("var")
```

`value` contains the value of the variable, or `nil`, if no data is available.
`status` contains the variable status message in case of error.
`"var"` is the variable name in data logger.

Function mstep.ReadInt

Reads from the data logger a variable of 32-bit integer type.

Usage:

```
value, status = mstep.ReadLong("var")
```

`value` contains the value of the variable, or `nil`, if no data is available.
`status` contains the variable status message in case of error.
`"var"` is the variable name in data logger.

Function mstep.ReadFloat

Reads from the data logger a variable of 32-bit floating point type.

Usage:

```
value, status = mstep.ReadLong("var")
```

`value` contains the value of the variable, or `nil`, if no data is available.
`status` contains the variable status message in case of error.
`"var"` is the variable name in data logger.

Function mstep.ReadString

Reads from the data logger a variable of string type.

Usage:

```
value, status = mstep.ReadString("var")
```

`value` contains the value of the variable, or `nil`, if no data is available.
`status` contains the variable status message in case of error.
`"var"` is the variable name in data logger.

Function mstep.WriteChar

Writes a value to data logger into a variable of 1-byte character type.

Usage:

```
status = mstep.WriteChar("var","A")
```

`status` contains the variable status message in case of error.
`"var"` is the variable name in data logger.
`"A"` is the new value of the variable.

Function mstep.WriteInt

Writes a value to data logger into a variable of 16-bit integer type.

Usage:

```
status = mstep.WriteInt("var",123)
```

`status` contains the variable status message in case of error.
`"var"` is the variable name in data logger.
`123` is the new value of the variable.

Function mstep.WriteLong

Writes a value to data logger into a variable of 32-bit integer type.

Usage:

```
status = mstep.WriteLong("var",123)
```

`status` contains the variable status message in case of error.
`"var"` is the variable name in data logger.
`123` is the new value of the variable.

Function mstep.WriteFloat

Writes a value to data logger into a variable of 32-bit floating point number type.

Usage:

```
status = mstep.WriteFloat("var",123.456)
```

`status` contains the variable status message in case of error.
`"var"` is the variable name in data logger.
`123.456` is the new value of the variable.

Function mstep.WriteString

Writes a value to data logger into a variable of string type.

Usage:

```
status = mstep.WriteString("var","text")
```

`status` contains the variable status message in case of error.
`"var"` is the variable name in data logger.
`"text"` is the new value of the variable.

Function mstep.logDebug

Writes a message to debug message. The path of the log file is the following:

```
/home/user/picolo4modules/picolo4modules.log
```

Usage:

```
mstep.logDebug("debug message")
```

`"debug message"` is the message to write.

Function mstep.registerCallback

Registers an event callback to be executed when an event with specified identifier occurs in data logger.

Usage:

```
status = mstep.registerCallback("2", "eventCallback2")
```

`status` contains the status message of registration in case of error or `nil` if no error occurred.
`"2"` is the event identifier in the data logger. Note: in firmware version 1.8 only numeric values stored as text are supported yet. The number is the zero-based index in the list of events in the data logger configuration.
`"eventCallback2"` is the name of a Lua function which will execute it the event occurs.

Function mstep.unregisterCallback

Removes a registration of callback(s) from the specified event identifier. If the same callback was registered to other events, this registration remains unchanged.

Usage:

```
status = mstep.unregisterCallback("2", "eventCallback2")
status = mstep.unregisterCallback("2", "eventCallback2", "otherCallback")
```

`status` contains the status message of deregistration in case of error or `nil` if no error occurred.
`"2"` is the event identifier in the data logger. Note: in firmware version 1.8 only numeric values stored as text are supported yet. The number is the zero-based index in the list of events in the data logger configuration.

"eventCallback2" is the name of a Lua function which will stop being executed in case of the event. "otherCallback" is the name of other Lua function to deregister. It is possible to deregister unlimited number of event handlers of same event. Simply add more parameters with callback names.

3. C extension Modules to Lua

The functionality of Lua can be extended by dynamic loading of library written in any programming language, which supports Lua. Only functions of the following prototype are supported:

```
int lua_CFunction (lua_State *L);
```

The arguments and return values must be programmed over the Lua Stack. See the Lua documentation for details. The return value is the number of returned parameters on stack. It is easy to create a wrapper function to an existing library.

Compile C modules with compiler: arm-linux-gnueabihf-gcc

Flags: -I"path to lua.h" -llua5.3.5 -fPIC

The modules need to be stored near the picolo4back application.

To use the module, add Require call to the beginning of Lua script. Example:

```
waves = require("libwaves_lua")
```

`libwaves_lua` is the name of the C extension module.
`waves` is the namespace, where the functions are loaded

Then the functions of the module are accessible by calling them in namespace.function() form. Example:

```
waves.updateWaveMeasurement(value)
```