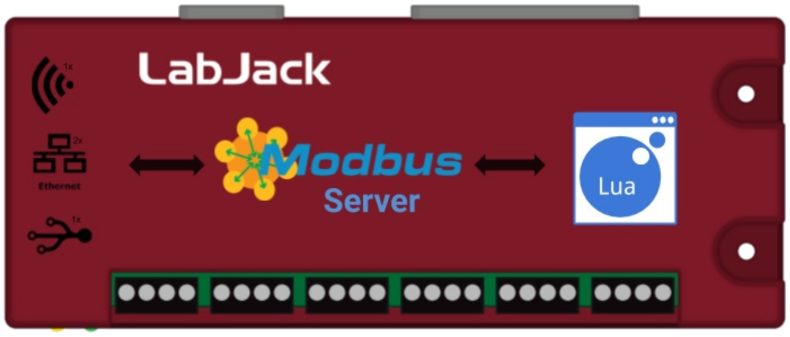
Published on *LabJack* (https://labjack.com)

25.0 Lua Scripting [T-Series Datasheet]

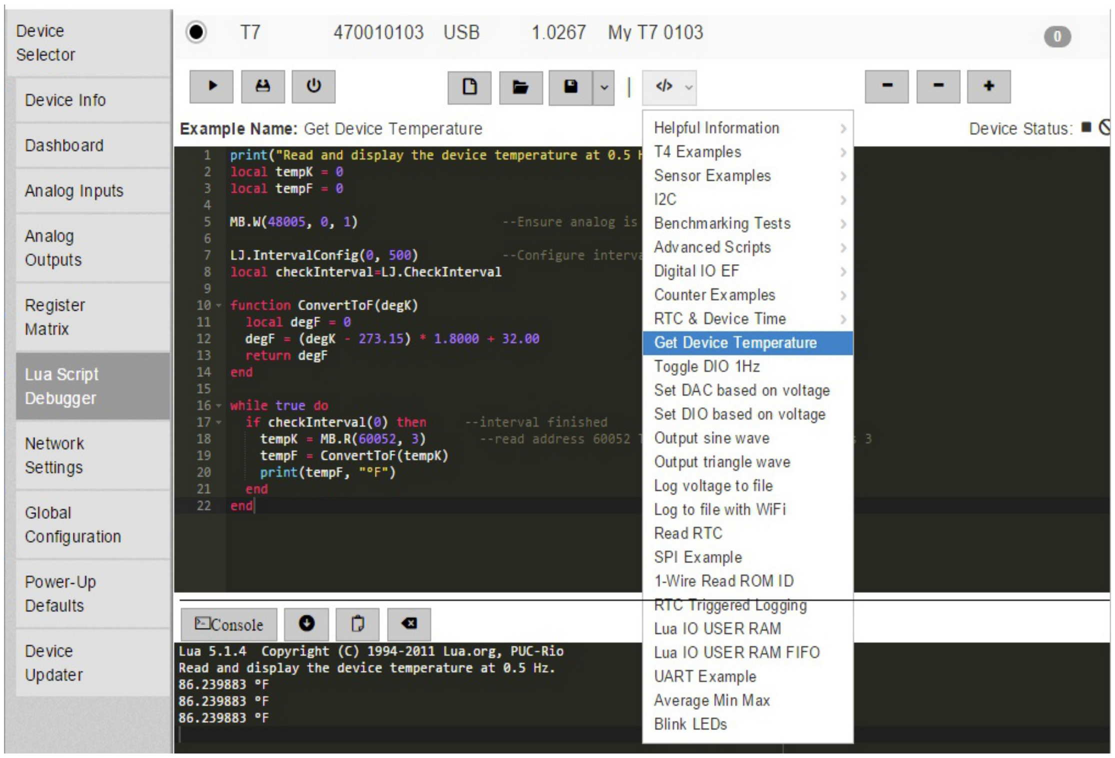
Lua Scripting Overview

T-Series devices can execute Lua code to allow independent operation. A Lua script can be used to collect data without a host computer or be used to perform complex tasks producing simple results that a host can read. For a good overview on the capabilities of scripting, see this LabJack Lua blog post.



Getting Started

1. Connect your device to your computer, launch Kipling, and navigate to the Lua Script Debugger tab.



Example Name: Get Device Temperature

```
print("Read and display the device temperature at 0.5 H
local tempK = 0
local tempF = 0

MB.W(48005, 0, 1)                        --Ensure analog is

LJ.IntervalConfig(0, 500)                --Configure interva
local checkInterval=LJ.CheckInterval

function ConvertToF(degK)
  local degF = 0
  degF = (degK - 273.15) * 1.8000 + 32.00
  return degF
end

while true do
  if checkInterval(0) then     --interval finished
    tempK = MB.R(60052, 3)       --read address 60052 T
    tempF = ConvertToF(tempK)
    print(tempF, "°F")
  end
end
```

```
Lua 5.1.4  Copyright (C) 1994-2011 Lua.org, PUC-Rio
Read and display the device temperature at 0.5 Hz.
86.239883 °F
86.239883 °F
86.239883 °F
```

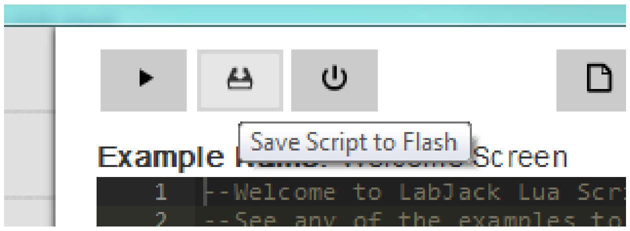
2. Open the "Get Device Temperature" example and click the Run button. The console should show the current device temperature. Now click the Stop button.

3. Try out some other examples.

Running a script when the device powers up

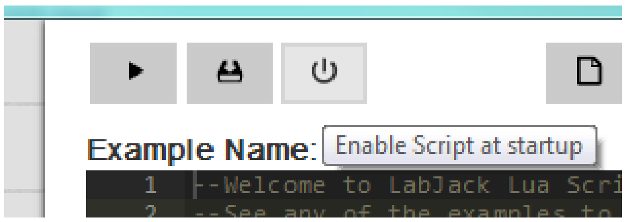
A T-Series device can be configured to run a script when it powers on or resets. Typically you should test scripts for a while with the Run/Stop buttons while viewing the debug output in the console. Then once everything is working correctly, enable the script at startup and close Kipling.

To enable the script at startup:



```
--Welcome to LabJack Lua Scr
--See any of the examples to
```

1. Click Save Script to Flash.



```
--Welcome to LabJack Lua Scri
--See any of the examples to
```

2. Click Enable Script at Startup. Now when the device is powered on or reset, it will run your script.

3. After power cycling the device, if it becomes un-usable and the COMM/Status lights are constantly blinking the Lua Script is likely causing the device to enter an invalid state. The device can be fixed by connecting a jumper wire between the SPC terminal and either FIO0 or FIO4, see the SPC section of the datasheet for more details.

A short video tutorial describing how to do this is available on theScreen Casting and Lua Script Tutorials news post.

Learning more about Lua

Learning Lua is very easy. There are good tutorials onLua.org as well as on several other independent sites. If you are familiar with the basics of programming, such as loops and functions, then you should be able to get going just by looking at the examples.

Lua for T-Series Devices

Try to keep names short. String length directly affects execution speed and code size.

Use local variables instead of global variables (it's faster and less error-prone). For example, use:

```
local a = 10
```

instead of:

```
a = 10
```

You can also assign a function to a local for further optimization:

```
local locW = MB.W
locW(2003, 0, 1) --Write to address 2003
```

Lua supports multi-return. Here, both table and error are returned values:

```
table, error = MB.RA(Address, dataType, nValues)
```

Freeing Lua memory

Since Lua occupies system RAM, it's good to clean up a Lua script when it's no longer needed. When a Lua script ends, by default it does not free memory used by the Lua. The following methods clean up Lua memory:

- Press the stop button in the Lua script tab in Kipling
- Write 0 to LUA_RUN
- Write 0 to LUA_RUN as the last command of your Lua script

For more memory cleaning options, see the system RAM section.

Limitations of Lua on T-Series Devices

Lua on the T4 and T7 has several limitations:

Speed: Maximum data rates can only be achieved with a host computer. Lua can not handle as much data, but is not limited by the communication overhead that a host computer is. The lack of overhead means that Lua can respond more quickly. Several benchmarking scripts are available in the Lua example set. The analog input benchmark runs at about 12.5kHz. The DIO benchmark runs at about 16kHz.

Data Types: On T-Series devices, Lua's only numeric data type is IEEE 754 single precision (float). This is more important than it sounds. Here is a good article on floating point numbers and their pitfalls: Floating Point Numbers. The single precision data type means that working with 32-bit values requires extra consideration. See the Working with 32-bit Values section below.

Script size: The T4 and T7 have 64 kBytes of RAM available. The Lua virtual machine requires about 25 kBytes. Script code adds memory requirements from there. When memory starts to get full, there are two places that are likely to throw errors:

- The first is when a Lua script is loaded and started, but has not yet started running. When a script is started, the source code is transferred to the device and that code is compiled into byte code. Memory for the compilation process can be freed up by reducing comments and the lengths of variables and functions.
- The second place memory errors can occur is while the script is running. Memory can be freed up using the collectgarbage function. Collecting prevents garbage from building up as much. If more memory is still needed, the script needs to be simplified or shrunk.

Notice: From a practical standpoint, Lua Scripts will start becoming too long and throwing out-of-ram errors after around 150 lines (depending on how long each line of code is). Once this limitation is encountered, there are a few tricks that can be used to implement additional features. They all involve making your code less readable but will assist in implementing additional features.

1. Remove any comments from your code as these consume precious RAM.
2. Edit the lua script file using an editor outside of Kipling and upload a minified version of the script instead of the full script file. The suggested tool to use for minifying Lua Scripts is here: https://mothereff.in/lua-minifier.

Lua libraries

Most of the Lua 5.1 libraries are available, with the exception of functions that rely on a host operating systems, such as Time and Networking.

There are some LabJack-specific libraries:

- I2C Library: Provides functions which simplify and reduce the memory requirements of scripts that use I2C.
- Bit Library: Provides bitwise functions such as AND, OR, NOR, and XOR.
- LabJack Library: Provides control of script timing and access hardware features of the LabJack device.

Passing data into/out of Lua

User RAM consists of a list of Modbus addresses where data can be sent to and read from a Lua script. Lua writes to the Modbus registers, and then a host device can read that information.

There are a total of 200 registers of pre-allocated RAM, which is split into several groups so that users may access it conveniently with different data types.

Use the following USER_RAM registers to store information:

User RAM Registers

Name	Start Address	Type	Access
USER_RAM#(0:39)_F32 Generic RAM registers. Useful for passing data between a host computer and a Lua script. Will not return an error if alternate data types are used.	46000	FLOAT32	R/W
USER_RAM#(0:9)_I32 Generic RAM registers. Useful for passing data between a host computer and a Lua script. Will not return an error if alternate data types are used.	46080	INT32	R/W
USER_RAM#(0:39)_U32 Generic RAM registers. Useful for passing data between a host computer and a Lua script. Will not return an error if alternate data types are used.	46100	UINT32	R/W
USER_RAM#(0:19)_U16 Generic RAM registers. Useful for passing data between a host computer and a Lua script. Will not return an error if alternate data types are used.	46180	UINT16	R/W

&print=true

USER_RAM example script:

```
while true do
 if LJ.CheckInterval(0) then
  Enable = MB.R(46000, 3) --host may disable portion of the script
  if Enable >= 1 then
   val = val + 1
   print("New value:", val)
```

```
      MB.W(46002, 3, val)  --provide a new value to host
    end
  end
end
```

There is also a more advanced system for passing data to/from a Lua script referred to as FIFO buffers. These buffers are useful if you want to send an array of information in sequence to/from a Lua script. Usually 2 buffers are used for each endpoint, one buffer dedicated for each communication direction (read and write). For example, a host may write new data for the Lua script into FIFO0, then once the script reads the data out of that buffer, it responds by writing data into FIFO1, and then the host may read the data out of FIFO1.

User RAM FIFO Registers - Advanced

Name	Start Address	Type	Access
USER_RAM_FIFO#(0:3)_DATA_U16 Generic FIFO buffer. Useful for passing ORDERED or SEQUENTIAL data between various endpoints, such as between a host and a Lua script. Use up to 4 FIFO buffers simultaneously->1 of each data type, all 4 different data types, or a mixture. e.g. FIFO0_DATA_U16 points to the same memory as other FIFO0 registers, such that there are a total of 4 memory blocks: FIFO0, FIFO1, FIFO2 and FIFO3. It is possible to write into a FIFO buffer using a different datatype than is being used to read out of it. This register is a buffer. Underrun behavior - throws an error.	47000	UINT16	R/W
USER_RAM_FIFO#(0:3)_DATA_U32 Generic FIFO buffer. Useful for passing ORDERED or SEQUENTIAL data between various endpoints, such as between a host and a Lua script. Use up to 4 FIFO buffers simultaneously->1 of each data type, all 4 different data types, or a mixture. e.g. FIFO0_DATA_U16 points to the same memory as other FIFO0 registers, such that there are a total of 4 memory blocks: FIFO0, FIFO1, FIFO2 and FIFO3. It is possible to write into a FIFO buffer using a different datatype than is being used to read out of it. This register is a buffer. Underrun behavior - throws an error.	47010	UINT32	R/W
USER_RAM_FIFO#(0:3)_DATA_I32 Generic FIFO buffer. Useful for passing ORDERED or SEQUENTIAL data between various endpoints, such as between a host and a Lua script. Use up to 4 FIFO buffers simultaneously->1 of each data type, all 4 different data types, or a mixture. e.g. FIFO0_DATA_U16 points to the same memory as other FIFO0 registers, such that there are a total of 4 memory blocks: FIFO0, FIFO1, FIFO2 and FIFO3. It is possible to write into a	47020	INT32	R/W

Name	Start Address	Type	Access
FIFO buffer using a different datatype than is being used to read out of it. This register is a buffer. Underrun behavior - throws an error.			
USER_RAM_FIFO#(0:3)_DATA_F32 Generic FIFO buffer. Useful for passing ORDERED or SEQUENTIAL data between various endpoints, such as between a host and a Lua script. Use up to 4 FIFO buffers simultaneously->1 of each data type, all 4 different data types, or a mixture. e.g. FIFO0_DATA_U16 points to the same memory as other FIFO0 registers, such that there are a total of 4 memory blocks: FIFO0, FIFO1, FIFO2 and FIFO3. It is possible to write into a FIFO buffer using a different datatype than is being used to read out of it. This register is a buffer. Underrun behavior - throws an error.	47030	FLOAT32	R/W
USER_RAM_FIFO#(0:3)_ALLOCATE_NUM_BYTES Allocate memory for a FIFO buffer. Number of bytes should be sufficient to store users max transfer array size. Note that FLOAT32, INT32, and UINT32 require 4 bytes per value, and UINT16 require 2 bytes per value. Maximum size is limited by available memory. Care should be taken to conserve enough memory for other operations such as AIN_EF, Lua, Stream etc.	47900	UINT32	R/W
USER_RAM_FIFO#(0:3)_NUM_BYTES_IN_FIFO Poll this register to see when new data is available/ready. Each read of the FIFO buffer decreases this value, and each write to the FIFO buffer increases this value. At any point in time, the following equation holds: Nbytes = Nwritten - Nread.	47910	UINT32	R
USER_RAM_FIFO#(0:3)_EMPTY Write any value to this register to efficiently empty, flush, or otherwise clear data from the FIFO.	47930	UINT32	W

&print=true

USER_RAM_FIFO example script:

```
aF32_Out= {}  --array of 5 values(floats)
aF32_Out[1] = 10.0
aF32_Out[2] = 20.1
aF32_Out[3] = 30.2
aF32_Out[4] = 40.3
aF32_Out[5] = 50.4

aF32_In = {}
numValuesFIO0 = 5
ValueSizeInBytes = 4
numBytesAllocFIFO0 = numValuesFIO0*ValueSizeInBytes
MB.W(47900, 1, numBytesAllocFIFO0) --allocate USER_RAM_FIFO0_NUM_BYTES_IN_FIFO to 20 bytes
```

```
LJ.IntervalConfig(0, 2000)
while true do
  if LJ.CheckInterval(0) then
    --write out to the host with FIFO0
    for i=1, numValuesFIO0 do
      ValOutOfLua = aF32_Out[i]
      numBytesFIFO0 = MB.R(47910, 1)
      if (numBytesFIFO0 < numBytesAllocFIFO0) then
        MB.W(47030, 3, ValOutOfLua)  --provide a new array to host
        print ("Next Value FIFO0: ", ValOutOfLua)
      else
        print ("FIFO0 buffer is full.")
      end
    end
    --read in new data from the host with FIFO1
    --Note that an external computer must have previously written to FIFO1
    numBytesFIFO1 = MB.R(47912, 1) --USER_RAM_FIFO1_NUM_BYTES_IN_FIFO
    if (numBytesFIFO1 == 0) then
      print ("FIFO1 buffer is empty.")
    end
    for i=1, ((numBytesFIFO1+1)/ValueSizeInBytes) do
      ValIntoLua = MB.R(47032, 3)
      aF32_In[i] = ValIntoLua
      print ("Next Value FIFO1: ", ValIntoLua)
    end
  end
end
```

Working with 32-bit Values

Lua is using a single precision float for its data-type. This means that working with 32-bit integer registers is difficult (see examples below). If any integer exceeds 24-bits (including sign), the lower bits will be lost. The workaround is to access the Modbus register using two numbers, each 16-bits. Lua can specify the data type for the register being written, so if you are expecting a large number that will not fit in a float (>24 bits), such as a MAC address, then read or write the value as a series of 16-bit integers.

If you expect the value to be counting up or down, use `MB.RA` or `MB.RW` to access the U32 as a contiguous set of 4 bytes.

If the value isn't going to increment (e.g. the MAC address) it is permissible to read it in two separate packets using `MB.R`.

Read a 32-bit register

```
--If value is expected to be changing and is >24 bits: Use MB.RA
aU32[1] = 0x00
aU32[2] = 0x00
aU32, error = MB.RA(3000, 0, 2)   --DIO0_EF_READ_A. Type is 0 instead of 1
```

```
DIO0_EF_READ_A_MSW = aU32[1]
DIO0_EF_READ_A_LSW = aU32[2]

--If value is constant and is >16,777,216 (24 bits): Use MB.R twice
--Read ETHERNET_MAC (address 60020)
MAC_MSW = MB.R(60020, 0)  --Read upper 16 bits. Type is 0 instead of 1
MAC_LSW = MB.R(60021, 0)  --Read lower 16 bits.

--If value is <16,777,216 (24 bits): Use MB.R
--Read AIN0_EF_INDEX (address 9000)
AIN0_index = MB.R(9000, 1)  --Type can be 1, since the value will be smaller than 24 bits.
```

Write a 32-bit register

```
--If value might be changed or incremented by the T7 and is >24 bits: Use MB.WA
aU32[1] = 0xFF2A
aU32[2] = 0xFB5F
error = MB.WA(44300, 0, 2, aU32) --Write DIO0_EF_VALUE_A. Type is 0 instead of 1

--If value is constant and is >24 bits: Use MB.W twice
MB.W(44300, 0, 0xFF2A) --Write upper 16 bits. Type is 0 instead of 1
MB.W(44301, 0, 0xFB5F) --Write lower 16 bits.

--If value is <16,777,216 (24 bits): Use MB.W
--Write DIO0_EF_INDEX (address 44100)
MB.W(44100, 1, 7)  --Type can be 1, since the value(7) is smaller than 24 bits.
```

Loading a Lua Script to a T7 Manually

Load Lua Script Manually To Device

While Kipling handles Lua scripting details easily and automatically, the example below shows how to load a Lua script to a T7 manually. The general process as well as some example pseudocode is below:

1. Define or load a Lua Script and make sure a device has been opened.
2. Make sure there is a null-character at the end of the string.
3. Make sure a Lua Script is not currently running. If one is, stop it and wait for it to be stopped.
4. Write to the "LUA_SOURCE_SIZE" and "LUA_SOURCE_WRITE" registers to instruct the T7 to allocate space for a script and to transfer it to the device.
5. (Optional) Enable debugging.
6. Instruct the T-Series device to run the loaded Lua Script.

The C example below opens a T7, shuts down any Lua script that may be running, loads the script, and runs the script.

```
const char * luaScript =
    "LJ.IntervalConfig(0, 500)\n"
    "while true do\n"
    "  if LJ.CheckInterval(0) then\n"
    "    print(LJ.Tick())\n"
    "  end\n"
    "end\n"
    "\0";

const unsigned scriptLength = strlen(luaScript) + 1;
// strlen does not include the null-terminating character, so we add 1
// byte to include it.

int handle = OpenOrDie(LJM_dtT7, LJM_ctANY, "LJM_idANY");

// Disable a running script by writing 0 to LUA_RUN twice
WriteNameOrDie(handle, "LUA_RUN", 0);
// Wait for the Lua VM to shut down (and some T7 firmware versions need
// a longer time to shut down than others):
MillisecondSleep(600);
WriteNameOrDie(handle, "LUA_RUN", 0);

// Write the size and the Lua Script to the device
WriteNameOrDie(handle, "LUA_SOURCE_SIZE", scriptLength);
WriteNameByteArrayOrDie(handle, "LUA_SOURCE_WRITE", scriptLength, luaScript);

// Start the script with debug output enabled
WriteNameOrDie(handle, "LUA_DEBUG_ENABLE", 1);
WriteNameOrDie(handle, "LUA_DEBUG_ENABLE_DEFAULT", 1);
WriteNameOrDie(handle, "LUA_RUN", 1);
```

The above example is valid C code, where the following functions are error-handling functions that cause the program to exit if an error occurs:

- OpenOrDie wraps LJM_Open
- WriteNameOrDie wraps LJM_eWriteName
- WriteNameByteArrayOrDie wraps LJM_eWriteNameByteArray

The Lua script in the example above is in the form of a C-string, e.g. a string with a null-terminator as the last byte.

To download a version of the above example, seeutilities/lua_script_basic.c, which includes a function that reads debug data from the T7.

Reading Debug Data/Print Statements

After a script has started (with debugging enabled) any information printed by a lua script can be read by a user application using the "LUA_DEBUG_NUM_BYTES" and "LUA_DEBUG_DATA" registers.

25.1 I2C Library [T-Series Datasheet]

Log in or register to post comments

I2C Library Overview

T7 firmware minimum: 1.0225

The I2C library abstracts most of the Modbus calls needed to run I2C. The abstraction allows users to focus on I2C rather than Modbus, and reduces the memory requirements of scripts. Several I2C examples can be found on the I2C Sensor Examples page.

I2C.config

```
Error = I2C.config(SDA, SCL, Speed, Options, Address)
```

Sets parameters that are not normally changed. Values set by this function will remain unchanged until this function is called again or the equivalent Modbus registers are written to.

Parameters:

- `SDA` - DIO pin # that will be used as the I2C data line
- `SCL` - DIO pin # that will be used as the I2C clock line
- `Speed` - See I2C documentation
- `Options` - See I2C documentation
- `Address` - Left Justified

Returns:

- `Error` - standard LabJack T-Series error codes.

I2C.writeRead

```
RxData, Error = I2C.writeRead(TxData, NumToRead)
```

This function will first write the data in `TxData` to the preset address, then will read `NumToRead` bytes from that same address.

Parameters:

- `TxData` - This is a Lua table containing the values to be transmitted. The size of the table determines the number of bytes that will be transmitted.
- `NumToRead` - The number of data bytes to be read.

Returns:

- `RxData` - A Lua table of the values read
- `Error` - standard LabJack T-Series error codes.

I2C.read

```
RxData, Error = I2C.read(NumToRead)
```

This function will read `NumToRead` bytes from the preset address.

Parameters:

- `NumToRead` - The number of data bytes to be read.

Returns:

- `RxData` - A Lua table of the values read
- `Error` - standard LabJack T-Series error codes.

I2C.write

```
Error = I2C.write(TxData)
```

This function will write the data in `TxData` to the preset address.

Parameters:

- `TxData` - This is a Lua table containing the values to be transmitted. The size of the table determines the number of bytes that will be transmitted.

Returns:

- `Error` - standard LabJack T-Series error codes.

I2C.search

```
AddressList, Error = I2C.search(FirstAddress, LastAddress)
```

This function will scan the I2C bus addresses are acknowledged. An acknowledged address means that at least one device is set to that address. Addresses are tested sequentially between the first and last address parameters.

Parameters:

- `FirstAddress` - The first address to be tested.
- `LastAddress` - The last address to be tested.

Returns:

- `AddressList` - A Lua table containing all the addresses that responded.
- `Error` - standard LabJack T-Series error codes.

25.2 Bit Library

Log in or register to post comments

Overview

Lua in T-Series devices is based on Lua 5.1.4 which did not include a bitwise library. A subset of Lua 5.2's Bitwise Library have been added. The name of the library is `bin` or `bit` instead of `bit32`. To use the AND function use `bin.band` instead of `bit32.band`.

Limitations on T-Series

The T4 and T7 use 32-bit floating point (single precision) numbers. To perform bitwise operations the 32-bit float is converted into an integer. The bitwise operation is performed. Then the integer is converted back into a floating point number. When converting to an integer some information can be lost. Any decimal places will be truncated off and only up to 23-bits will make it to the integer. This is because 8 bits is used for the exponent and 1 bit for sign.

Functions

Operation of the below functions match the Lua 5.2 bitwise library with the exception of the data type limitations.

- `arshift`
- `band`
- `bnot`
- `bor`
- `bxor`
- `lshift`
- `rshift`

25.3 LabJack Library

Log in or register to post comments

LabJack's Lua library

The `MB` and `LJ` libraries allow access to the Modbus map and provide timing control.

Modbus Address Functions

The following Modbus address functions are similar to eReadAddress, eWriteAddress, eReadAddressArray, and eWriteAddressArray. The address and data type need to be provided. The address controls what you want to read or write. The data type controls how the data should be interpreted. Address and data types can be found in the LabJack Modbus Map and in the Kipling Register Matrix.

Data Types - Below is a list of data type indices. The data types match those used by the LJM Library.

- **0** - unsigned 16-bit integer
- **1** - unsigned 32-bit integer
- **2** - signed 32-bit integer
- **3** - single precision floating point (float)
- **98** - string
- **99** - byte - The "byte" dataType is used to pass arrays of bytes in what Lua calls tables

MB.R

```
value, error = MB.R(Address, dataType)
```

Modbus read. Reads a single value from a Modbus register. The type can be a u16, u32, a float, or a string. Any errors encountered will be returned in error.

MB.W

```
error = MB.W(Address, dataType, value)
```

Modbus write. Writes a single value to a Modbus register. The type can be a u16, u32, a float, or a string. Any errors encountered will be returned in error.

MB.WA

```
error = MB.WA(Address, dataType, nValues, table)
```

Modbus write array. Reads `nValues` from the supplied table, interprets them according to the `dataType` and writes them as an array to the register specified by `Address`. The table must be indexed with numbers from 1 to `nValues`.

MB.RA

```
table, error = MB.RA(Address, dataType, nValues)
```

Modbus read array. Reads `nValues` of type `dataType` from `Address` and returns the results in a Lua table. The table is indexed from 1 to `nValues`.

Shortcut Functions:

The following functions are shortcuts used to gain a small speed advantage over the equivalent Modbus functions.

LJ.ledtog

`LJ.ledtog()` --Toggles status LED. Note that reading AINs also toggles the status LED.

LJ.Tick

`Ticks = LJ.Tick()` --Reads the core timer (1/2 core frequency).

LJ.DIO_D_W

`LJ.DIO_D_W(3, 1)` --Quickly change FIO3 direction _D_ to output.

LJ.DIO_S_W

`LJ.DIO_S_W(3, 0)` --Quickly change the state _S_ of FIO3 to 0 (output low)

LJ.DIO_S_R

`state = LJ.DIO_S_R(3)` -- Quickly read the state _S_ of FIO3

LJ.CheckFileFlag

`flag = LJ.CheckFileFlag()` and `LJ.ClearFileFlag()`

`LJ.CheckFileFlag` and `LJ.ClearFileFlag` work together to provide an easy way to tell a Lua script to switch files. This is useful for applications that require continuous logging in a Lua script and on-demand file access from a host. Since files cannot be opened simultaneously by a Lua script and a host, the Lua script must first close the active file if the host wants to read file contents. The host writes a value of 1 to FILE_IO_LUA_SWITCH_FILE, and the Lua script is setup to poll this parameter using `LJ.CheckFileFlag`. If the file flag is set, Lua code should switch files:

Example:

```
fg = LJ.CheckFileFlag() --poll the flag every few seconds
if fg == 1 then
  NumFn = NumFn + 1               --increment filename
  Filename = Filepre..string.format("%02d", NumFn)..Filesuf
  f:close()
  LJ.ClearFileFlag()             --inform host that previous file is available.
  f = io.open(Filename, "w")      --create or replace a new file
  print ("Command issued by host to create new file")
end
```

Timing Functions:

LJ.IntervalConfig & LJ.CheckInterval

`LJ.IntervalConfig` and `LJ.CheckInterval` work together to make an easy-to-use timing function. Set the desired interval time with `IntervalConfig`, then use `CheckInterval` to watch for timeouts. The interval period will have some jitter but no overall error. Jitter is typically ±30 µs but can be greater depending on processor loading. A small amount of error is induced when the processor's core speed is changed.

Up to 8 different intervals can be active at a time.

```
LJ.IntervalConfig(handle, time_ms)
```

Sets an interval timer, starting from the current time.

`handle`: 0-7. Identifies an interval.

`time_ms`: Number of milliseconds per interval.

```
timed_out = LJ.CheckInterval(handle)
```

`handle`: 0-7. Identifies an interval.

Returns: 1 if the interval has expired. 0 if not.

Example:

```
LJ.IntervalConfig(0, 1000)
while true do
  if LJ.CheckInterval(0) then
    --Code to run once per second here.
  end
end
```

Lua Performance Functions:

LJ.setLuaThrottle

```
LJ.setLuaThrottle(newThrottle)
```

Set the throttle setting. This controls Lua's processor priority. Value is number of Lua instruction to execute before releasing control to the normal polling loop. After the loop completes Lua will be given processor time again.

LJ.getLuaThrottle

```
ThrottleSetting = LJ.getLuaThrottle()
```

Reads the current throttle setting.