Lua Scripting Overview

> Lua scripting is an advanced topic. eGauge Support cannot review code and has limited support for troubleshooting Lua scripts.

Introduction

Support for eGauge Script (eScript) was introduced in firmware v1.1, allowing limited scripting through the use of formula registers. eScript was designed for calculating basic and common mathematical formulas such basic arithmetic operations, comparisons, and conditionals, along with the ability to call library routines (functions).

Firmware v4.1 introduced Lua v5.3 support. To ensure safe operation, Lua execution is guarded with timeouts to protect against infinite loops or unacceptably long execution times. Additionally, a restricted set of standard libraries is supported.

To maintain backwards compatibility, firmware transparently translates eScript expressions to Lua scripts before execution and the Lua execution environment provides all the functions available within eScript.

Lua Primer

Lexical Conventions

Names may consist of any letters, digits, or underscores but may not start with a digit. Numeric values use the same conventions as in most other languages, including the ability to write hexadecimal values with a prefix of **`0x`**. The two boolean literal values are **`true`** and **`false`**. A variable with an undefined value is **`nil`**. Strings may be enclosed in double or single quotes. Characters within a string may be escaped with a backslash. Comments start with a double-dash (**`--`**) and extend to the end of the line. Statements may be terminated with a semicolon but, generally, semicolons are not required (even when there are multiple statements on a single line). The only exception to this rule is that a semicolon is required if the following statement starts with a left parenthesis.

Variables

Variables are *global* unless explicitly declared as local. For example:

```
x = 10       -- global variable
local y = 42   -- local variable
```

Operators

Lua provides a normal set of operators for arithmetic operations and comparisons. A bit unusual is that the inequality operator is **`~=`** rather than the more typical **`!=`**. Logical operators use keywords like in Python:

and, **or**, and **not**. The length of a string or a table (see Structured Data) is obtained by prefixing the string or table name with a hash mark. **#'hi'** would return 2, for example. Strings can be concatenated with the double-dot operator. **'yell'..'ow'** would yield **'yellow'**, for example.

Control Structures

The syntax for the various control structures is:

```
for index=initial,step,final do block end -- numeric for loop
```

```
for var in iterator do block end   -- iterator for loop
```

```
while cond do block end    -- while loop
```

```
repeat block until cond    -- do while loop
```

```
if cond then block else block end  -- conditional statement
```

You can use **break** to exit a loop but, unlike in C and other languages, there is no `continue` that would allow you to skip to the next iteration.

Structured Data

Lua uses tables to represent both arrays and hash tables (Python dictionaries, Javascript objects). For example:

```
local a = {'a', 'b', 'c'}
```

assigns an array containing the strings **'a'**, **'b'**, and **'c'** to local variable **a**. Unlike in most other languages, the array base index is 1, so **a[1]** would yield the first element or **'a'** in our example.

Dictionary literal values can be written like this:

```
local d = {['k1']='v1', ['k2']='v2'}
```

and indexed with square brackets such that **`d['k2']`** would yield **`'v2'`**, for example. If the keys for a dictionary happen to be valid Lua names, the square brackets and quotes around the key strings can be omitted. For example, the above example could be simplified to:

```
local d = {k1='v1', k2='v2'}
```

For such key values, it is also possible to access their values using member-name syntax. For example, **`d.k1`** would yield **`'v1'`**, just as **`d['k1']`** would.

Migrating from eScript to Lua

Most of eScript has a direct equivalent in Lua. eScript has full support for a single numeric type (IEEE754 double precision float) and limited support for strings. As configured for the eGauge firmware, Lua has the same numeric type but also supports 32-bit integers and has full support for strings, boolean values, and tables.

The most important differences between eScript and Lua are as follows:

- In eScript, the value of a register is obtained with **`$"register_name"`**, whereas in Lua, the equivalent expression is **`__r("register_name")`**.
- Lua boolean values cannot directly be used as numeric values, whereas eScript uses 0 to represent false and any non-zero value for true.
- Lua does not provide a direct analog for the conditional operator

```
cond ? if_true_expr : if_false_expr
```

  Instead, Lua uses the logical expression

```
cond and if_true_expr or if_false_expr
```

  This works quite similarly to the eScript conditional because of the way the **`and`** and **`or`** operators are defined. Specifically, **`and`** returns **`false`** if the left-hand side is **`nil`** or **`false`** or the right-hand side's value otherwise. Operator **`or`** returns the value of the left-hand side if it is not **`nil`** or **`false`** and the value of the right-hand side otherwise. Both operators short-circuit evaluation and operator **`and`** has higher precedence than **`or`**.

- eScript automatically propagates NaN (Not-a-Number) values. For example, if any function is called with a NaN value, the returned result is also NaN. Similarly, if the condition of the

conditional operator is NaN, then the result of the conditional expression is also NaN.

Given the similarities between eScript and Lua, most eScript expressions trivially translate to Lua. The non-trivial translations are shown in the table below:

eScript expression	**Lua equivalent**
`a < b`	`__lt(a, b)`
`a <= b`	`__le(a, b)`
`a > b`	`__gt(a,b)`
`a >= b`	`__ge(a,b)`
`a = b`	`__eq(a,b)`

`c ? a : b`	`(function(__c)`
	`  if __c ~= __c then return __c end`
	`  return __c~=0 and (a) or (b)`
	` end)(c)`

In words, the comparison operators get translated to calls to helper functions **`__lt()`**, **`__le()`** and so on. These helper functions check if either **a** or **b** is a NaN and return NaN, if so. If not, they perform the comparison and return 0 for false and 1 for true. The translation of the conditional operator is more complicated because care has to be taken to handle NaN properly and to evaluate **a** and **b** only when necessary. This is accomplished in Lua with the help of an anonymous inline function which checks whether the condition is NaN and returns NaN if that is the case. Otherwise, the function checks if the condition has a non-zero value and, if so, returns the value of **a**. Otherwise, it returns the value of **b**.

Lua Environment provided by eGauge Firmware

Standard Environment

For safety reasons, the eGauge firmware provides a restricted Lua environment (the Lua sandbox). Basic functions and variables are limited to the following subset (see Lua 5.3 manual for a detailed description):

```
_VERSION, assert, error, getmetatable, ipairs, load, next, pairs, pcall,
print, rawequal, rawget, rawlen, rawset, select, setmetatable, tonumber,
tostring, type, xpcall
```

The following standard Lua libraries are available:

string, math, table

Additions Provided by eGauge Firmware

Basic and Alert functions

All functions available to eScript are also available to Lua scripts. See the online documentation of an eGauge meter for a complete list (**Help** ? **Basic functions** or **Help** ? **Alert functions**). When these functions are called from Lua, they will also propagate NaN values, just like for eScript. That is, if any of the functions are called with a NaN argument, the return value will also be NaN.

Module **`json`**

This module provides the ability to encode a Lua object to a string and safely convert a string back to an object again.

`string` = **`json.encode`**(*`value`*):

This function accepts a Lua value and serializes it to the corresponding JSON string, which it returns. Tables that contain cyclical references cannot be JSON-encoded and will result in an error. The maximum size of the JSON-encoded string is currently limited to 4095 bytes. Lua tables may be indexed by a mix of number, boolean, and string values, whereas JSON objects are always indexed by strings. This function converts Lua tables that are not empty and whose indices consist entirely of numbers in the range from 1 to *N* (where *N* is the length of the table) to JSON arrays and everything else to JSON objects. In the latter case, numeric and boolean indices are converted to their equivalent strings.

`value` = **`json.decode`**(*`string`*):

This function accepts a string and deserializes it to the corresponding Lua object. Only double-quotes are allowed for string quoting. White space consisting of blanks or tabs is ignored.

Module **`persistent`**

This module provides variables whose values persist across power outages and device restarts.

`obj` = `persistent:new(`*`name, initial, description`*`)`:

Declares a persistent variable with the specified *`name`*. Unlike Lua names, this name can be an arbitrary string. The name must be unique as any Lua script declaring a persistent variable of the same name will access the same underlying object. If this is the first time the persistent variable has been declared, its value is set to *`initial`*. The purpose of the variable must be described by the string passed for *`description`*. The return value is an object representing the persistent variable.

`obj``:get()`

Returns the current value of the persistent variable represented by object *`obj`*.

`obj``:set(`*`value`*`)`

Sets the value of the persistent variable represented by object *obj* to *value*. Any value acceptable to **json.encode()** may be set.

Lua Environment for Control Scripts

Control scripts have access to the standard environment available as described in the previous section. They also have access to all basic and alert functions as well as to the coroutine library. Several low-level functions as well as several convenience modules are available as well, as described below.

Low-level Functions

Most of these functions are normally not used directly. They provide the low-level mechanisms required to implement the higher-level abstractions provided by the modules described in the sections below.

tid, *err* = __ctrl_submit(*attrs*, *method*, *args*…)

Submit a call to the method named *method* on the device identified by *attrs*, passing arguments *args*. The method name must be a string, *attrs* a table of name/value pairs, and *args* must be a sequence of zero or more Lua argument values that are compatible with the argument types expected by the named method. The method name may be a fully qualified method name consisting of an interface name, immediately followed by a dot (.), immediately followed by the proper method name or a proper method name on its own. In the latter case, the method is invoked through the first interface registered for the device that implements the named method. Otherwise, the method is invoked through the named interface.

__ctrl_submit returns two values: a transaction id *tid* and an error string *err*. On success, *tid* is a non-negative number which uniquely identifies the newly created call object and *err* is nil. On error, *tid* is a negative error code (see ctrl.Error below) and *err* is an optional string that may provide an explanation of why the call failed. The error string, if provided, is typically translated to the locale set on the meter.

status, *result* = __ctrl_result(*tid*):

Get the result for the method call identified by transaction id *tid*. The *tid* must be non-negative and must have been returned by a previous call to __ctrl_submit.

__ctrl_result returns two values: an integer *status* and *result*. The *status* is zero on success, in which case *result* is the result returned by the method call, converted to a Lua value. On error, *status* is a negative error code (see ctrl.Error below) and *result* is nil. In particular, if a method call is still in progress, error code ctrl.Error.AGAIN (-8) is returned. In this case, the caller should wait for a little bit and then retry the __ctrl_result call again until it succeeds.

status, *err* = __ctrl_cancel(*tid*):

Attempt to cancel the method call identified by transaction id *tid*. This *tid* must be non-negative and must have been returned by a previous call to __ctrl_submit.

`__ctrl_cancel` returns two values: an integer *`status`* and an optional error string err. The *`status`* is zero on success, in which case *`tid`* is guaranteed to be an invalid transaction id, until it is reused and returned by another call to `__ctrl_submit`. On error, *`status`* is a negative error code (see `ctrl.Error` below) and *`err`* is an optional string that may provide an explanation of why the call failed. The error string, if provided, is typically translated to the locale set on the meter.

`ret``, `*`err`*` = __ctrl_get_devices(`*`attrs`*`)`:
Get a list of devices that match the optional attributes specified by table *`attrs`*. If the *`attrs`* is omitted or `nil`, a list of all known (registered) devices is returned.

`__ctrl_get_devices` returns two values: table *`ret`* and an optional error string *`err`*. On success, *`ret`* is a list of tables and *`err`* is `nil`. Each table in the returned list corresponds to a matching device in no particular order. The table contains the name/value pairs registered for that device.

On error, *`ret`* is `nil` and *`err`* may be a non-`nil` string explaining what went wrong, typically translated to the locale set on the meter.

`ret`` = __ctrl_get_interface(`*`name`*`)`:
Get a particular interface or a list of all known interfaces. Argument *name* must be a string that names the desired interface or omitted or `nil` to get a list of all interfaces.

`__ctrl_get_interface` returns a single value *`ret`*. If there was an error or if the requested interface cannot be found, `nil` is returned. Otherwise, *`ret`* is a single interface (if *name* was specified) or a list of interfaces (if *name* was omitted or `nil`). Each interface is described by a table with the following members:

- `name`: The name of the interface as a string.
- `methods`: A list of methods implemented by the interface.

Each method is described by a table with the following members:

- `name`: The name of the method as a string.
- `arg_types`: The DBus type signature of the arguments as a string.
- `ret_type`: The DBus type signature of the return value as a string.
- `doc`: A description of what the method does as a string. This string may contain references to the arguments which are enclosed in `<arg>`/`</arg>` tags. For example, the string "`<arg>foo</arg>`" would refer to the method argument named "`foo`".
- `arg_names`: A string of comma-separated argument names, in order of appearance. This is used only for documentation purposes as, other than in the `doc` string, the argument names have no significance. For example, the string "`foo,bar`" would indicate that the method expects two arguments, which are referred to in the `doc` string as argument names `foo` and `bar`, respectively.

ret = __sleep(*time*):

Suspend execution of the call for *time* seconds. The specified time may be a fractional amount, not just an integer amount. If the Lua program calling __sleep has other runnable coroutines, the other coroutines are executed. If there are no runnable coroutines left, execution of the program is suspended for the minimum amount of time required until the first coroutine becomes runnable again.

The function returns an integer *ret* which is zero on success or negative if an error occurred.

ret = sleep(*time*):

This is an alias for __sleep and may be used by end-user Lua programs as a convenience. Libraries should always call __sleep instead to ensure the intended function is executed even if a Lua program redefines the name sleep to another value.

Module **ctrl**

This module provides a higher-level interface to invoke control methods. It is generally preferable to use this module rather than the low-level functions documented by the previous section.

dev = ctrl:dev(*attrs*, *obj*):

Create a control device object with attributes *attrs*. The optional *obj* argument could be specified to implement an extended control device class, but is usually omitted (or you could pass **nil** to it) to create a new object. This operation does not communicate with the remote device identified by *attrs* and therefore returns immediately.

iface = *dev*:interface(*name*):

Create an interface object for the interface identified by *name* of device *dev*. The returned interface object will contain a proxy method for each method defined by the named interface. The proxy methods can be called like any other method and automatically forward the call to the control device and then wait for the result to become available from the device. As such, these operations can take a long time to complete and will call __sleep() as needed. Because of this, other coroutines may be executed while a proxy method call is in progress.

status, *result* = *dev*:call(*method*, *args…*)

Call the method named *method* on device *dev*, passing arguments *args* to it and return the result. The method name must be a string and *args* must be a sequence of zero or more Lua argument values that are compatible with the argument types expected by the named method. The method name may be a fully qualified method name consisting of an interface name, immediately followed by a dot (.), immediately followed by the proper method name or a proper method name on its own. In the latter case, the method is invoked through the first interface registered for the device that implements the named method. Otherwise, the method is invoked through the named interface.

Two values are returned: an integer *`status`* and *`result`*. The *`status`* is zero on success, in which case *`result`* is the result returned by the named method, converted to a Lua value. On error, *`status`* is a negative error code (see `ctrl.Error` below) and *`result`* is `nil`.

`Error`:
: This table declares symbolic names for various control errors, namely:

- `UNSPEC` (-1): An unspecified error occurred.
- `INVAL` (-2): An invalid or incompatible argument was passed to a method.
- `NODEV` (-3): The device attributes specified an invalid device path.
- `ATTRS` (-4): The attributes do not match the selected device.
- `NOMETHOD` (-5): The method name could not be found.
- `NOENT` (-6): The specified transaction id could not be found.
- `BUSY` (-7): The device is busy (too many pending calls).
- `AGAIN` (-8): The call is still pending.

Module **`tasks`**

This module provides a convenient interface for creating several tasks that may be run quasi concurrently and then executing them until they have all completed.

`tasks:add(`*`fun, args`*`…)`:
: Add a task which, when executed, runs function *`fun`* with arguments *`args`* until the function returns. The function may call `__sleep` or `coroutine.yield` to suspend execution temporarily and give other tasks a chance to execute.

`tasks:run()`:
: Execute previously added tasks until they have all completed.
