eTactica Plugin Development Kit

The eTactica device support software, (mlifter) is capable of being extended to support new Modbus meters using simple Lua plugins.

Target Audience

Writing plugins is straightforward, but this is not a task end users would normally ever contemplate. You need a good understanding of Modbus, and at least basic programming skills, and at least basic understanding of Lua The existing plugins should provide very good templates for creating your own device plugin. This document targets both internal and external software developers.

Required Information/Equipment

You need at *least* the modbus tables for the device you want to support. The meter vendor's website is the most likely place to find this, but you may have to contact the vendor directly. You will almost definitely need an eTactica Gateway and the target device, and to have it wired correctly.

Basic Information

The Plugin Admin page in the EG web console provides full management of the plugins, but if you prefer to work manually, the following paths are used behind the scenes

Plugin type	storage
user	/etc/remake.d/plugins_device
system	/usr/lib/lua/remake/plugins_device

All non-disabled plugins are loaded at startup, and every "allowed for auto" plugin is used when probing devices. User provided plugins are used first, to ensure that if you upload an updated version of any plugin that it takes precedence. (If a modbus device was configured with either a category or an explicit plugin, then only those matching plugins are used for probing)

Every plugin runs in it's own sandboxed Lua environment

Every plugin is provided a "state" table where it can preserve variables/state between device reads. (Typically, for storing scaling factors or phase configurations that only need to be read once)

A command line tool is provided to test running these plugins (plugin_tool) and in recent versions, an online UI is provided for editing and testing plugins too.

"Plugins" - Administration page at the admin console

The Plugins administration page at the admin console hosts a full adminstration interface to all existing plugins on the gate. It enables the user to

- Upload/download lua plugin files
- Download lua plugins files from the gate
- View and edit source code of the lua plugins
- Enable/Disable plugins, and flag "automatic" or not

Walkthrough

A lua plugin needs to provide at least two functions and one variable

- function probe(state, slaveid) *or* probe_static(state, slaveid)
- function read(state)
- variable plugin_type

Additionally you should add the following function

- function get_description()

The following two functions are optional, but can allow automatic scaling, config and control of individual variables

- function config()
- function variables()

A plugin is provided with a range of modbus functions,

- read_registers(base_address, count)
- read_input_registers(base_address, count)
- write_register(address, value)
- write_registers(address, values)
- report_slave_id()

Variables plugins must provide

plugin_type

This variable must be declared in your file. It must contain a string from the following table.

Water/gas/environment plugins all use the same data format, the "type" here is used in configuration, and to make it easier to probe for known devices

plugin_type value	Meaning	
electricity	Plugin will be available for selection in menus of electricity meters, and automatic probing within families	
water	As electricity but for water flow type meters	
gas		
environment	Targetted at plugins that are collecting weather/temp type readings	

If you don't want to remember the strings, the utilities module provides some defines that can be used, eg

```
local rmep = require("remake.plugin_support")
plugin_type = rmep.PLUGIN_TYPE.WATER
plugin_type = "water"  -- equivalent
```

Functions plugins must provide

probe(state, slaveid)

This function is called to test whether this plugin supports a given modbus slave. The plugin should make any tests necessary to be confident that the unknown modbus slave is supported by this plugin. This is important, if a plugin indicates that it supports a slave device, no other plugin will be used. Probe functions should robustly check that the unknown modbus device is indeed the expected sort, and not simply return true blindly here. If you cannot reliably detect your device, you should *not* provide this function, and instead provide probe_static

Arguments

- `state` This is a private table provided to this plugin, its state is kept between calls to this plugin. You can use this to save information about product differences that might be important to save doing multiple modbus reads at a later date. Or, you can completely ignore the table altogether.
- `slave_id` This is the (numeric) Modbus slave id of the device that is being probed. This can be helpful for verifying the device. Some Modbus devices place this information at specific known table offsets.

Return values

Probe is expected to return two variables,

- `success`. true if this plugin can be used to read this device, false otherwise
- `result`. if not successful, an error code can be provided. Otherwise, this should be a table of data results

The probed table of data results should look something like this:

probe result table

```
{
 -- REQUIRED FIELDS
    -- the identifier for this device, ideally a serial number, should
at
    -- least be unique for this gateway
    id = "P3s234F002",

    -- OPTIONAL FIELDS

    -- information about the software running on the device, all five
fields are optional
    -- The software information is presented to the user and can be
helpful for verification, but is not used for
    -- anything at this point. Defaults to 0.0.false  See GATE-301
    major_version = 2,
    minor_version = 7,
    -- dirty indicates a development build, if this is relevant
    dirty = false,

 -- these are free text fields used in the display
 vendor_name = "Sample Vendor Text"
 product_name = "eTactica EM"
 }
```

Electricity plugin optional fields

Field	Description
phase_count	how many phases are configured for this meter. Defaults to 3 if not specified, Clamped to the range of 1,2,3
reactive	whether the meter can measure reactive power (VAR Hours) Defaults to true.

probe_static(state, slaveid)

If this method is provided instead of probe, then the plugin will not be eligible for automatic probing, but will still be available for explicit configuration.

read(state)

If a plugin returns true from the probe call (or probe_static), its read method will be called periodically to collect readings. The probe function will *not* be called again. If your device could potentially reset, you should detect this in your read routine and call probe again yourself.

Arguments

The state parameter is the same private, persistent state table given to the probe function.

Return values

As in probe, the return value is two values,

Read is expected to return two variables,

- success. true or false, true if everything succeeded as expected. False for modbus read failures, timeouts, invalid data, phase error detection, etc
- result. if not successful, an error code can be provided. Otherwise, this should be a table of data results
  If not noted specifically, values not provided will default to 0

The data table can be in one of two formats. The original "Electricity" style or the "Generic" style.

Electricity plugin data results

electricity read result table

```
{
 -- per phase decimal amps
 amp_1 = 45.60,
 amp_2 = 3.9,
 amp_3 = 12.312,
 -- per phase decimal volts
 volt_1 = 229.320,
 volt_2 = 228.970,
 volt_3 = 229.111,
 -- per phase power factors, signed, limited to the range of -1 to +1
    -- negative numbers indicate generation, postive consumption
 phase_1 = 0.98,
 phase_2 = -0.41,
 phase_3 = 0.97,
 -- frequency in decimal Hertz.  Defaults to 50 if not provided
 frequency = 49.98,
 -- valid phases can be used to indicate that no valid readings are
available for this phase,
 -- (power factor < 0.4 or voltage sag below 70V, or similar)
 -- This defaults to true, but can be explicitly marked as false if
desired.
    -- invalid phases are not published in any messages
 --valid_phase_1 = true,
 --valid_phase_2 = false,
 --valid_phase_3 = true,
 -- watt hours and var hours, in decimal
 wh_cumulative = 1231241.234,
 -- This value is ignored if probe_result["reactive"] is false
 varh_cumulative = 3243241.324,
}
```

All other types of plugins

All other types of plugins use a slightly more verbose result table format, to allow for more arbitrary labeling and grouping. The table is instead a list of data point values, each including their unit and label. This format is based on SENML, also used in geras/hypercat and friends, and directly convertible to it. (The gateway wraps these individual elements up to make a senml packet)

```
{
    -- 'u' for unit is optional, but makes the display better
    -- label needs to be unique within this device
   e = {
    { n = "temperature_furnace_vent", v = 1349, u="C" },
    { n = "depth_tank_1", v = 4.568, u="m" },
    { n = "depth_tank_2", v = 5.2, u="m" },
    { n = "shipped_boxes", v = 345432 }
   }
}
```

For generic plugins, the final data point names become something of the form "<collection device id>/<probed device id>/<label from plugin>" Eg, for a gateway device with id = A840410012CB, and a generic plugin whose "probe" method returned an id of "1234532", the datapoints generated for the example code above would be

- A840410012CB/1234532/temperature_furnace_vent
- A840410012CB/1234532/depth_tank_1
- etc

Functions a pluign *should* provide

get_description()

Optional. The plugin returns a string (HTML Allowed) that should cover at least the following points:

- date when this plugin was written
- origin/author of plugin
- devices that this plugin supports
- known issues

Example: "Plugin for XYZ brand meters. Smart Developer, 06/2013. Supports all versions, all optional fields are provided."

Arguments

None.

Return values

A string containing the description.

variables()

Optional. Providing this function will allow individual variables to be able to be turned on or off individually, and scaled individually.

Arguments

None

Return values

An array of "variable" descriptions. Each entry contains the keys: "key", "description" and optionally "unit"

Example

```
function variables()
 local r = {
        { key="temp", unit="Cel", description="Internal temperature of
device" },
  { key="current/1", unit="A", description="Current on channel one" },
    }
 return r
end
```

config()

Optional. Providing a config function allows a plugin to be used in different ways, or to support different hardware variants for instance. Providing default values and validation information even allows the UI to present options to the user.

Arguments

None

Return values

An array of "config" items. Each entry contains the keys: "key", "description" and optionally, "defval", "type" and even a "validation" block. The "validation" block is passed virtually untouched to Knockout-Validation for use in the Plugin Adminstration pages and the Device Configuration pages. The "key" field is used for saving config options, all other values are only needed for the UI and presentation.

Example

```
function config()
    local v = {
        {
            key = "source_kwh_in/1",
            description = "Which obis key to use for the cumulative
kwh",
            defval = "1.8.0",
            type = "string",
            validation = {
                pattern = {
                    params = "^1.8.0$|^1.20.0$",
                    message = "Either 1.8.0 or 1.20.0"
                },
            },
        },
    }
    return v
end
```

Functions provided to plugins

read_registers(base_address, count) (Modbus function code 3 aka "Read Holding Registers")

This function is provided to the plugin, it is how a plugin's probe/read methods can do modbus register reads.

Arguments

`base_address` the integer address you wish to read from, eg 50000, 0x2000

`count` the number of **16bit** Modbus registers you wish to read from that base_address, eg 10, 42, 1

Return values

read_registers returns two values,

- success. true or false, true if everything succeeded as expected. False for modbus read failures, timeouts, invalid data, phase error detection, etc
- result. if not successful, an error code is provided, though there's very little you can do with this other than pass it on. Otherwise, this should be a table of data results

The data results are a flat table of registers, where the first index is the base_address register.

Example Modbus table in device

register offset	description	value
0x2000	voltage in volts	226
0x2001	current in amps	43
0x2002	frequency * 10	499

Example lua code using this function

```
result, registers = read_registers(0x2000, 3)
if result then
 print(string.format("Voltage is %d", registers[1]))
 print(string.format("Current is %d", registers[2]))
 print(string.format("Frequency is %f", registers[3] / 10))
else
 print("Failed to read from modbus device, code:" .. registers)
end
```

read_input_registers(base_address, count) (Modbus function code 4 aka "Read Input Registers")

This is virtually identical to "read_registers" but uses Modbus function code 4 instead of Modbus function code 3. The same arguments and return values are used. Support for this function was added in GATE-578 - plugins: add read_input_registers support RESOLVED (Since version 1.20 of the gate software)

write_register(address, new_value) (Modbus function code 6 aka "Write Single Register")

This is provided to plugins, but should only be used if you are *very* sure you are communicating with the correct device. Writing to registers of an unknown/misidentified devices could have potentially disastrous side effects. However, this functionality can be useful/desired and so is provided to plugins via GATE-555 (Since version 1.20 of the gate software)

Arguments:

address - register address to write

value - new value for that register (numeric, will be truncated to 16bit)

Return results:

write_register returns two values,

- success. true or false, true if everything succeeded as expected. False for modbus read failures, timeouts, unsupported operation or so on
- result. if not successful, an error code is provided, though there's very little you can do with this other than pass it on. For successes, this will be nil

write_registers(base_address, {new_values}) (Modbus function code 16 (0x10) aka "Write Multiple Register")

Very similar to write_register, but uses a different modbus function code, and writes a table of values, instead of a single value. The same warnings and cautions apply.

Arguments:

base_address - register address of first value to write.

new_values - An array style table of new values, applied sequentially from base_address

Return results:

write_registers returns two values,

- success. true or false, true if everything succeeded as expected. False for modbus read failures, timeouts, unsupported operation or so on
- result. if not successful, an error code is provided, though there's very little you can do with this other than pass it on. For successes, this will be nil

Example usage

```
-- Write 8 registers from 0x2010.  0x123456 will be written as 0x3456
local result, err = write_registers(0x2010, {12, 99, 0xbabe, 0xcafe, 0, 0x123456, 0x42, 0x69})
if not result then return false, err end
print("happily wrote multiple registers")
```

report_slave_id() (Modbus function code 17 (0x11) aka "Report Server Id")

This method is provided to plugins, to help with device identificaiton.

Arguments:

No arguments needed.

Return results:

report_slave_id returns two or three values,

- success. true or false, true if everything succeeded as expected. False for modbus read failures, timeouts, invalid data, phase error detection, etc
- result. if not successful, an error code is provided, though there's very little you can do with this other than pass it on. Otherwise, this should be a table of data results
- number of elements. if successful, number of elements is the number of elements in the table of the returned result. If not successful, this value is not set.

The data results table corresponds to the return value of the modbus function "report slave id" , only the first two bytes are not included (function code and byte count):

Description	Number of Elements	Value
Slave ID	1+ (device specific)	device specific, often not the actual slave id but a constant value.
Run Indicator Status	1	0x00 if OFF, 0xFF if ON
Additional Data	0+ (device specific)	device specific, often this part of the result is the most relevant.

Utilities module

A lua module is provides with the following helpful routines and constants

using the module

```
-- include this line near the top of your plugin
local rmep = require("remake.plugin_support)
```

Multi register reads

Routines for reading

What you need	helper function
read 32 bit signed	rmep.get_int32
read 32 bit unsigned	rmep.get_uint32
read 32bit word swapped	rmep.get_[u]int32_le
read 64bit signed	rmep.get_int64
read 64bit unsigned	rmep.get_uint64
read 32bit float	rmep.get_float32
read 32bit float word swapped	rmep.get_float32le

All of these routines take a table of registers, and an offset as their argument.

The unit test code below may be helpful

```
describe("Busted unit test example of get_int32 vs get_uint32 vs
get_uint32le vs get_int32le", function()
    it("should handle large postive and small negative properly",
function()
        -- sample register contents
        local regs = { 0xffff, 0xfffe }
        assert.are.equal(-2, rmep.get_int32(regs, 1))
        assert.are.equal(-65537, rmep.get_int32le(regs, 1))
        assert.are.equal(0xfffffffe, rmep.get_uint32(regs, 1))
        assert.are.equal(0xfffeffff, rmep.get_uint32le(regs, 1))
        assert.are.equal(4294967294, rmep.get_uint32(regs, 1))
    end)
end)
```

example usage in plugin

```
local rmep = require("remake.plugin_support)

function read(state)
 ....
 -- this reads two consecutive registers into a single signed 32bit
number.
 -- It follows proper most significant word semantics
 local amps = rmep.get_int32(registers, 3)
 ....
end
```

Return codes

A table of predefined return codes is provided, that *should* be used.

Code	Value	Meaning
RC.NO_ERROR	0	
RC.BAD_PARAMETER	1	read_registers will return this if you request 0 registers or pass invalid arguments
RC.MODBUS_CONNECTION_CLOSED	2	
RC.MODBUS_PROTOCOL	3	these two will be returned from various modbus failures
RC.UNKNOWN_ERROR	4	
RC.UNRECOGNISED	5	your plugin should return this if it doesn't support this device
RC.NO_SPI	6	Specific to the EM, normally indicates mains failure, *should not be used by 3rd party device plugins*
RC.ALL_PHASES_BAD	7	Returned when all three phases have power factors less than 0.5, normally indicates a meter that is improperly connected and should not be trusted
RC.NO_METER_REPLY	8	Used by "eye" (ER) type devices, that are indirectly reading a remote meter

Plugin Types

These values can be used in your plugin declaration if you prefer not to work with raw strings

Code	value
PLUGIN_TYPE.ELECTRICITY	electricity
PLUGIN_TYPE.WATER	water
PLUGIN_TYPE.GAS	gas
PLUGIN_TYPE.ENVIRONMENT	environment
PLUGIN_TYPE.INDIRECT	indirect

Example usage (from the Socomec plugin)

```
local rmep = require("remake.plugin_support")
plugin_type = rmep.PLUGIN_TYPE.ELECTRICITY
function probe(state, slave_id)
    -- All Socomec DIRIS A meters should have the slave id at this
location
    local result, register = read_registers(0xE123, 1)
    if not result then
        -- if read_registers failed, "register" contains the error
message
        return false, register
    end
    if (register[1] ~= slave_id) then
        return false, rmep.RC.UNRECOGNISED
    end
 .... more code
end
```

Testing a plugin

If you want test a plugin before uploading to your live EG, you should upload the lua file to a temporary directory on the EG, then SSH to the EG console. Use the **`plugin_tool`** application to test your plugin, like so.

Here we are using the bare minimum plugin attached at the bottom of this page.

```
root@eTactica_EG_F6F111:~# plugin_tool -f /usr/minimal.lua -s 0
2014-02-24T23:31:17 INFO plugin_device_handler.c: Successfully loaded
device support plugin: /usr/minimal.lua
2014-02-24T23:31:17 INFO modbus_support.c: Modbus connection localhost
is closed, attempting reconnection: 0
0 bytes flushed
2014-02-24T23:31:17 INFO plugin_device_reader.c: Found valid modbus
device using plugin: minimal.lua, serial: this is a fake device
Probed Device details dump:
 Modbus Slave ID: 0 (0)
 Device ID (serial or pseudo-serial): this is a fake device
 Vendor code was not provided.
 Vendor name was not provided.
 Product code was not provided.
 Product name was not provided.
 Phase Count: 3
 Reactive Power: true
 Version Information: Major: 0, Minor: 49, Dirty: no
root@eTactica_EG_F6F111:~
```

By default, plugin_tool only performs a probe, but there are other options to do reads, and dump extra verbose information.

```
root@eTactica_EG_F6F111:~# plugin_tool -h
plugin_tool is a tool for helping develop Lua plugins to support extra
energy meter devices.  It provides an environment identical to
the data collection software, and provides methods for running
the probe and read functions of your plugin.
For more information, see the eTactica Gateway Plugin Development
Kit documentation.
plugin_tool - usage:
  -d device, --device=device
   specify the Modbus/TCP device, eg localhost, 192.168.49.1
  -f pluginfile, --file=pluginfile
   specify the path for the lua plugin to test
  -s slave_id, --slave=slave_id
   specify the modbus slave id to test
  -r, --read
   Test the read functionality as well.
  -vXX, --verbose=XX specify a specific verbosity level (0..99)
  -v, --verbose specify generally verbose logging
  -h, --help  Print this help
root@eTactica_EG_F6F111:~#
```

Example Files

minimal-fake.lua works as far as being a valid plugin goes, but it provides fixed fake data.