Creating and Using Lua scripts

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

> Use caution and test extensively when using Lua scripts for automated control of remote devices to prevent unexpected operation or damage to connected equipment.

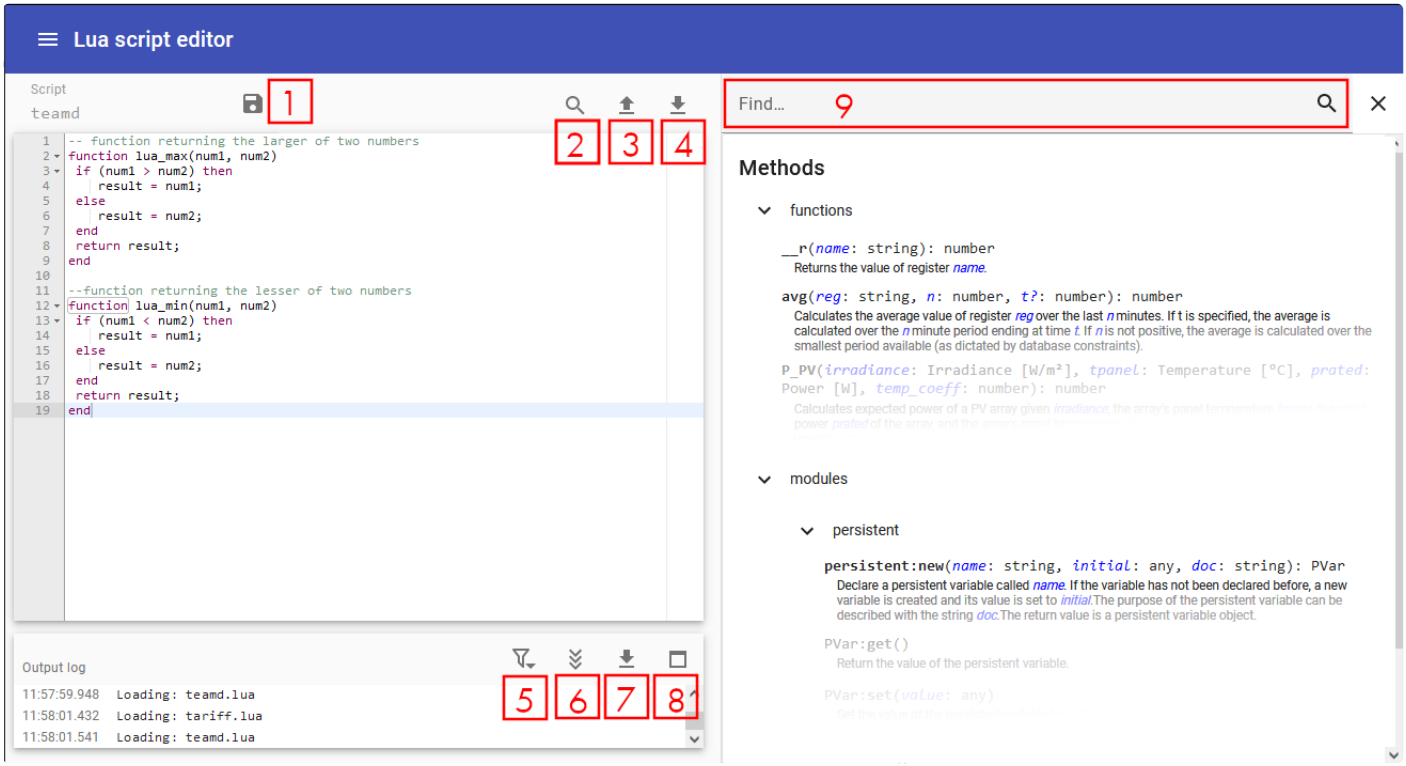
> **Use the latest firmware**: Lua support was added in eGauge meter firmware version 4.1. However, bug fixes and new features this article refers to may have been added in newer firmware versions. Click here for information about checking and updating a meter's firmware.

Introduction

eGage meters have a Lua scripting environment that allows for creation of advanced formula register and alert functions, as well as automated control of supported devices such as the eGauge PRM3 Power Relay Module.

Refer to the Lua Scripting Overview article for a general Lua primer and additional modules provided by the eGauge firmware.

Lua scripting interface



Several keyboard shortcuts are displayed when the interface is loaded:

- F1 : Open the keyboard shortcuts menu
- F2 : Toggle the sidebar on the right
- Ctrl-␣ : Toggle auto-complete
- Ctrl-, : Open the GUI settings menu

When using Mac, use the ⌘ key instead of Ctrl.

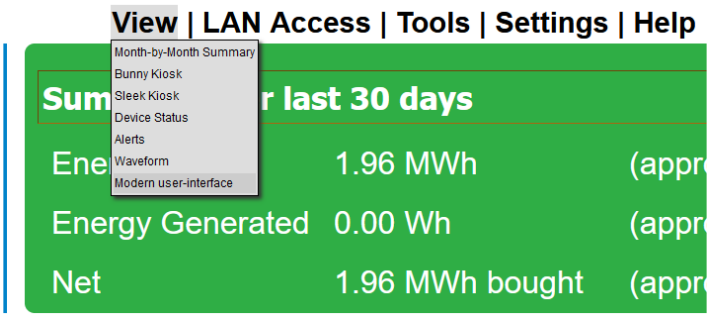
The eGauge web interface has a built-in Lua script editor with 3 panes as shown in the above screenshot.

- **Editor** (*top left*)— The Lua code editor. There are 4 buttons for this pane:
  - [1] Save: This saves the functions and makes them available for use by the meter.
  - [2] Search: This allows to search for and replace text in the code.
  - [3] Upload: This replaces the current script on the meter with a file upload.
  - [4] Download: This downloads the current script from the meter as a file
- **Output log** (*bottom left*)— Below the editor, the Lua log is displayed. This will include any script errors or print statements. If there are no custom Lua functions for the script loaded, there may be an error such as "Lua error: cannot open xxx.lua: No such file or directory", and this error may be ignored. There are 4 buttons available for this pane:
  - [5] Filter: Apply a filter to only output log lines containing certain text
  - [6] Autoscroll: Toggles auto-scrolling to the most recent log output
  - [7] Download: Downloads the log file
  - [8] Maximize Window: Maximizes the log output window
- **Sidebar** (*right*)— the sidebar on the right-hand side provides a method explorer to find and describe all available eGauge-provided functions and modules. The function or module may be clikced on to automatically insert it into the Editor pane to the left. The

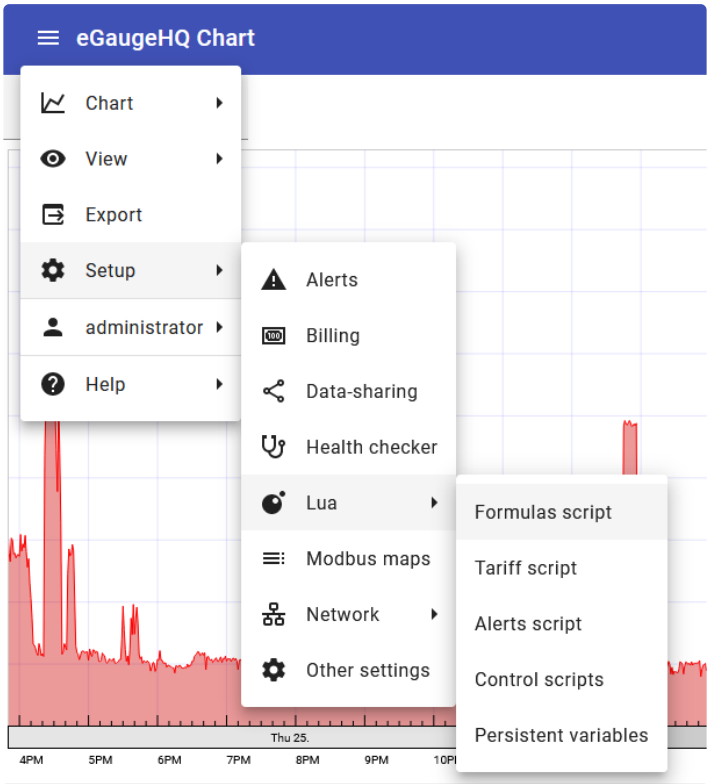
sidebar has a search bar [9] to conveniently search through methods, functions and documentation

Accessing the Lua scripting environment

1. If using the classic interface, click on View → Modern user-interface:



2. Navigate to Setup → Lua and choose the appropriate script type:



Script Types

Click to expand information for the script type:

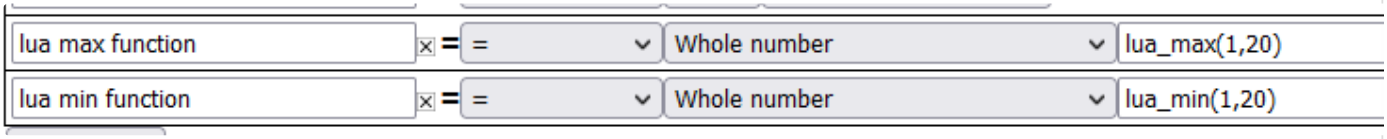
Formulas script (formula registers)

Creating lua functions in the Formulas script editor will allow the functions to be used in a formula register.

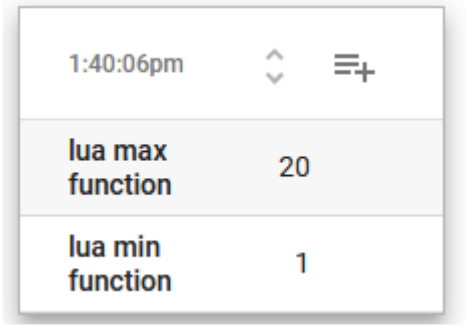
For example, here are two functions made in the Lua Formulas script editor that will return the min or max of two numbers:

```
-- function returning the larger of two numbers
function lua_max(num1, num2)
 if (num1 > num2) then
    result = num1;
 else
    result = num2;
 end
 return result;
end

--function returning the lesser of two numbers
function lua_min(num1, num2)
 if (num1 < num2) then
    result = num1;
 else
    result = num2;
 end
 return result;
end
```

They may then be used in a formula register:



And we may see the registers work as defined by the script functions:



Tariff script (Advanced time-of-use and tiered billing)

Advanced time-of-use and tiered billing may be performed in this Lua scrript.

A tariff script should provide (at least) a `cost(register, negate, schedule)` function which calculates the incremental cost based on the energy-use recorded by register. *negate* can be set to true if the register's value counts down for power consumption. *schedule* is optional and can be set to the (non-default) name of the schedule to use when calculating the cost.

A formula register of type "monetary" would be used with the `cost()` function.

For an example billing script, from the classic interface:

1. Navigate to Settings → Billing.
2. Choose Xcel Colorado as the tariff provider.
3. Click "OK" to save and go back to the main settings page.
4. Go back to Settings → Billing.
5. Change the tariff providert to "custom"
6. Click the "Customize tariff script" button, which will open a copy of the Xcel Colorado billing tariff in the Lua script editor.

Alert script

Alerts scripts work the same as Formula scripts, but are used in eGauge meter alerts, configured in Settings → Alerts.

Control scripts

> See the main Lua Scripting Overview Control Scripts section for additional Lua Control environment information.

> There is risk of damaging external equipment using control scripts. Only skilled Lua developers familiar with the eGauge meter and software should attempt to use Lua control scripts.

Control scripts can be used to confrol supported equipment such as the eGauge Power Relay Module (PRM3).

For example, the following script reads the instantaneous value of a register called "Temperature" and controls a PRM3 relay contact. If the temperature is lower than 21 C, relay number 0 of the PRM3 is closed (activated), otherwise opens (turns off) relay number 0. It then

sleeps for 15 minutes before checking again.

In the real world, the control script should be more advanced

```
dev = ctrl:dev({interface='relay'})
  relay = dev:interface('relay')

  while true do
    print("Temperature is currently: " .. __r("Temperature"))
    if __r("Temperature") < 21 then
       relay:close(0)
    else
       relay:open(0)
    end
    sleep(60*15)
  end
```

Persistent variables

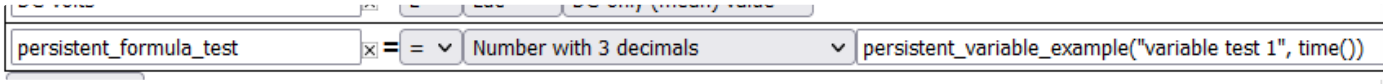
See the main Lua Scripting Overview section on Persistent variables.

Persistent variables are variables that are preserved between reboots or power cycles.

The following Formula script creates and updates a peristent variables with a given name and number passed to it in a formula register, and prints debug to the output log:

```
function persistent_variable_example(name, number)

    obj = persistent:new(name, number, "Variable stored by formula function persistent_variable_example")

    current_value = obj:get()
    print(name .. " currently has value " .. current_value)

    print("updating " .. name .. "to new value " .. number)

    obj:set(number)
  end
```

A register is configured to run the formula script:



This creates or updates a persistent variable called "variable test 1" with the current time.

The Formula script editor shows the print debug as the variable is updated once a second as the formula register is run:

```
15:26:28.168 variable test 1 currently has value 22.043333333333
15:26:28.168 updating variable test 1 to new value 22.043611111111
15:26:29.171 variable test 1 currently has value 22.043611111111
15:26:29.172 updating variable test 1 to new value 22.043888888889
15:26:30.168 variable test 1 currently has value 22.043888888889
15:26:30.168 updating variable test 1 to new value 22.044166666667
15:26:31.168 variable test 1 currently has value 22.044166666667
15:26:31.168 updating variable test 1 to new value 22.044444444444
```

While this particular example is rather pointless, persistent variables may be used in any Lua scripts. Control scripts are executed continuously, and a formula register would not need to be created to run it.
