NRIS Battery pack microcontroller - Design Document

1. Summary

The system is composed of a microcontroller development board (MCU dev board) interfaced with a predesigned PCB peripheral circuit board designed to monitor and control a battery pack. The dev board used was the Wemos d1 mini pro, which has an ESP8266 mcu. The PCB was also designed to use the ESP32 dev board (whose version I am not sure of). The PCB consists of;

- a circuit that uses 12V to power an external circuit with a 5V usb port,
- a GPIO controlled Relay,
- a push switch to control the Relay
- a circuit that uses 12V to power a 5V rail,
- a voltage divider circuit to convert 0 to 15V to the full range of the MCU ADC range.
- VE.direct serial port

With assistance and guidance from Derick Thiart, what was managed so far was soldering and testing all the necessary components of the PCB and configuring the microcontroller for the Relay and ADC. I have included some information on how to create a plugin, which is chosen to configure the VE.direct serial communication. A labelled diagram of the all detachable components to the PCB is shown in Figure 1, with their descriptions in table 2.

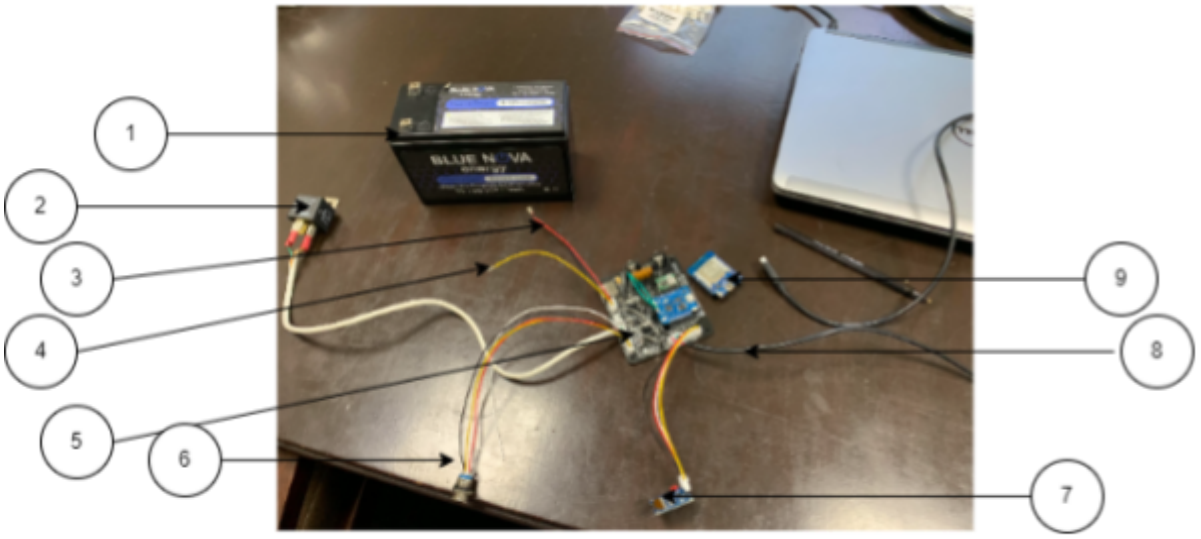


Figure 1: Labelled Diagram of all detachable components

Table 1: Descriptions of detachable components.

Label	Part	Description
1	12V Battery	
2	Relay	
3	12V power wires	Bothe the Red and Black wires are connected to 12V
4	GND wire	Note it is yellow, not black because of how the wire was packaged.
5	Main board with esp8266 development board	
6	LED switch	Note the white wire was disconnected and needs resoldering
7	OLED Display	
8	VE.direct cable	
9	ESP32 development board	

2. Soldering the PCB

Ordering Parts:
The PCB and schematics were already predesigned. The first task was to understand what the circuit was designed for and what parts were needed to complete the circuit. A parts list was created in order to identify and order missing parts.
When compiling the list, I noticed the schematics required the LM7805 linear voltage regulator while the L7805 voltage regulator from STMicroelectronics was already available. I verified, using the datasheets, that the two regulators are linear regulators with similar electrical characteristics and the same pin configuration.

Soldering:
It was recommended to first solder the small discrete components (such as capacitors, transistors and resistors) before soldering larger components (such as the darlington transistors and voltage regulators). All the parts were soldered except for the resistors needed for the ADC circuit and the DC Buck converter which had to be configured to output 5V when given 12V in. After designing the ADC circuit and configuring the buck converter, the Buck converter was soldered on while the ADC resistors were left to be soldered on until later.

Testing the Voltage Regulation:
Once the soldering was completed, I tested and verified that the USB port has 5V power to it. Continuity around the circuit was also tested.

ADC resistors:
According to the ESP8266ex datasheet, the pin for Analog to DC conversion expects 0 to 1V for a full 8 bit conversion. A voltage divider circuit was designed to convert 0-15V to the full range of the ADC. I added the requirement that the divider circuit should source a maximum of 1mA.
$R1/(R1 + R2)Vin = Vout$ (1)
$Vin/(R1 + R2) = 1mA$ (2)
I chose R1 to be 1kohms and R2 to be 15kohms resistors with 1% tolerance each.

3. ESPEasy Firmware

ESPEasy is a free open source IoT Firmware. In simple terms it provides a very simple operating system to the esp8266, where gpio pins and peripherals can be configured, and plugins and controllers act like applications. Firstly, I was assisted in uploading a build (binary file) of ESPEasy to test and understand the environment. Secondly, I had to compile my own binary file using PlatformIO on Visual studio Code. Instructions on Flashing a binary file and Compiling your own builds are given in the Appendices. Compiling the code took the longest as I needed a few admin permissions to update software on a computer with good processing power (Refer to appendix D for more details on system requirements).

The tasks needed to be done after uploading ESPEasy were:

1. Displaying Information on the OLED Display
2. Configuring the Relay and Switch
3. Setting up VE.Direct Serial communication port
4. Configuring the ADC scaling

Displaying Information on the OLED Display

Instructions on connecting with the OLED display can be found in appendix A. I was assisted in understanding how to communicate with the OLED display with the esp8266. A plugin on ESPEasy can be enabled to communicate with the OLED and command it what to display, at what refresh rate. I used an online resource https://emariete.com/en/connect-an-oled-display-ssd1306-to-espeasy/ to setup the OLED to Display the IP address it was connected to, the time of the day, the value measured by the ADC pin, and the state of the relay.

Configuring the Relay and Switch

The objective was to configure a GPIO pin as a switch that turns the main relay on and off. The switch should instantly turn the relay on, and the switch must be held in for 3 seconds to turn the relay off. I decided to use ESPEasy Rules to create the state sequence. Rules allow the user to ‘code’ instructions based on system events. Instructions on writing I Rules are given in Appendix C. I also decided to use a dummy device to store two variables to enable the logic to switch the relay on or off.

Configuring the ADC

A plugin for an ADC taking in an external signal was chosen. After designing resistor value and soldering them on. I tested the ADC maximum value on ESPEasy for a voltage at 15V, and used 15/max_value to scale the ADC value to its corresponding voltage value.

Setting up VE.Direct Serial communication port

A MPPT charge controller was to be used to communicate with the esp8266. First I studied documentation on VE.direct protocol and the Victron MPPT charge controller. Second I studied source code from github that used arduino microcontrollers that communicate with VE.direct. I then had to decide whether to build my own plugin or use a pre-existing plugin to receive VE.direct serial data. I was handed a BMV battery charge controller that also uses the VE.direct to test the plugins, while the MPPT was not present.

I did not find any plugin for VE.direct or a generic serial communication plugin that meets my needs. I decided to build my own plugin with a 4 Step approach.

1. Copy a stable existing plugin and rename it to see how plugins are added to ESPEasy (Successful)
2. Create a new plugin that toggles an LED (Successful)
3. Create a source file (separate from ESPEasy) that uses VE.direct to communicate with the BMV. (Successful)
4. Use all the previous steps and create a plugin to receive serial data from the BMV (Unsuccessful)

On step 3, I intended to use the USART with the usb cable to see if correct data was being read from the BMV. I realised that the esp8266 only has one serial port and I cannot use it for VE.direct and USART. I then decided to set up the OLED using the Adafruit library to receive debug information. Steps on using the Adafruit library can be seen in Appendix A.

Once I reached step 4, I had insufficient time to complete the plugin.

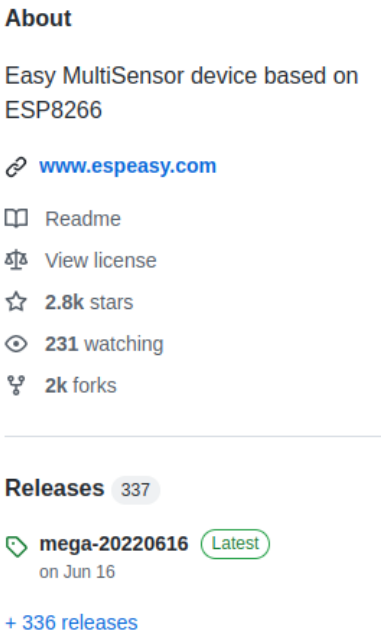
Appendix A

How to get started on ESPEasy with esp8266

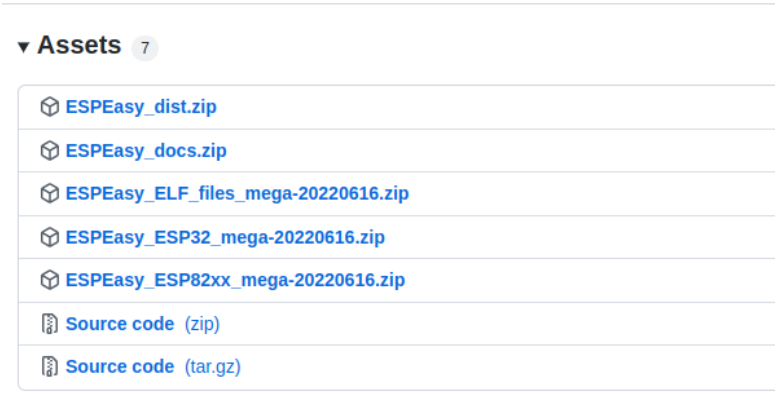
These instructions show how to flash a build of ESPEasy to a compatible esp device (where the esp8266 will be used as an example). More detailed instructions for esp82xx and esp32 can be found at the letscontrolit website, here.

Download the necessary folders

1. Go to the letscontrolit github repository, here, and below the releases tab select the latest build.



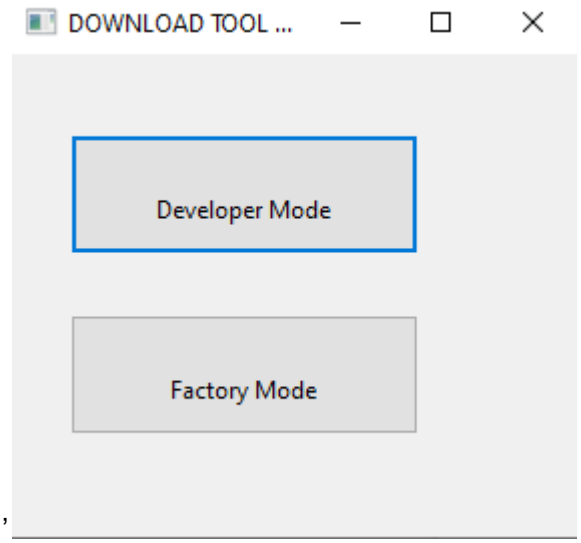
2. Under 'Assets' in the mega release you chose, download the ESPEasy_ESP82xx_mega-xxx.zip folder to your local computer. Choose any folder to save it to.



3. Unzip the folder to your location of choice.
   Then in the ESPEasy_ESP82xx_mega-xxxx file, go into Espressif_flash_download_tool_vX.X.X and run the flash_download_tool_vX.X.X.exe.

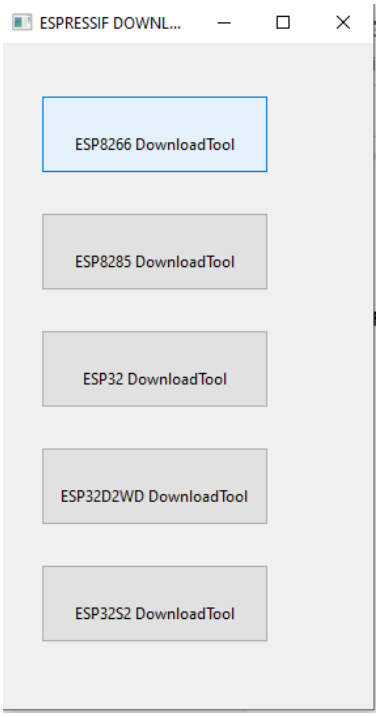
Flash to the mcu

1. When flash_download_tool_vX.X.X.exe is open, select ‘Developer Mode’

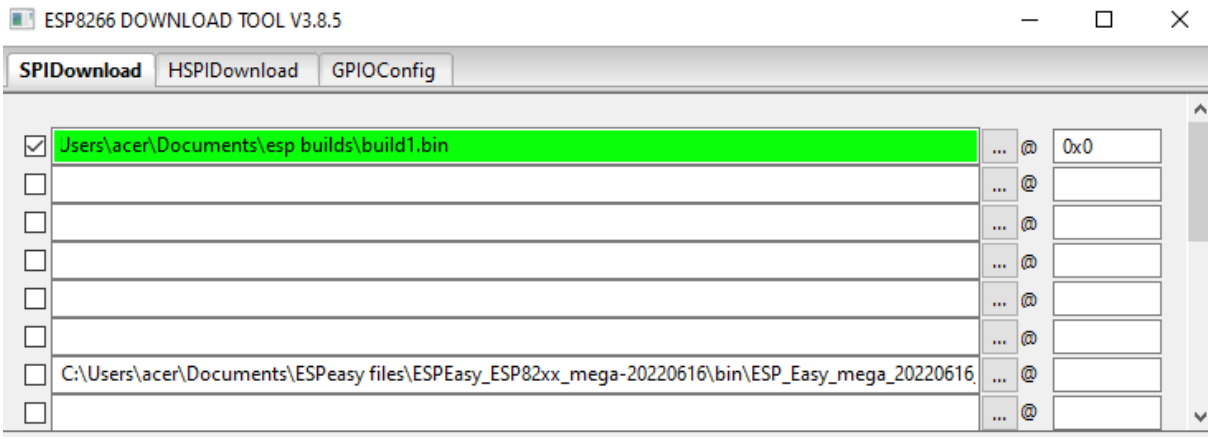


,

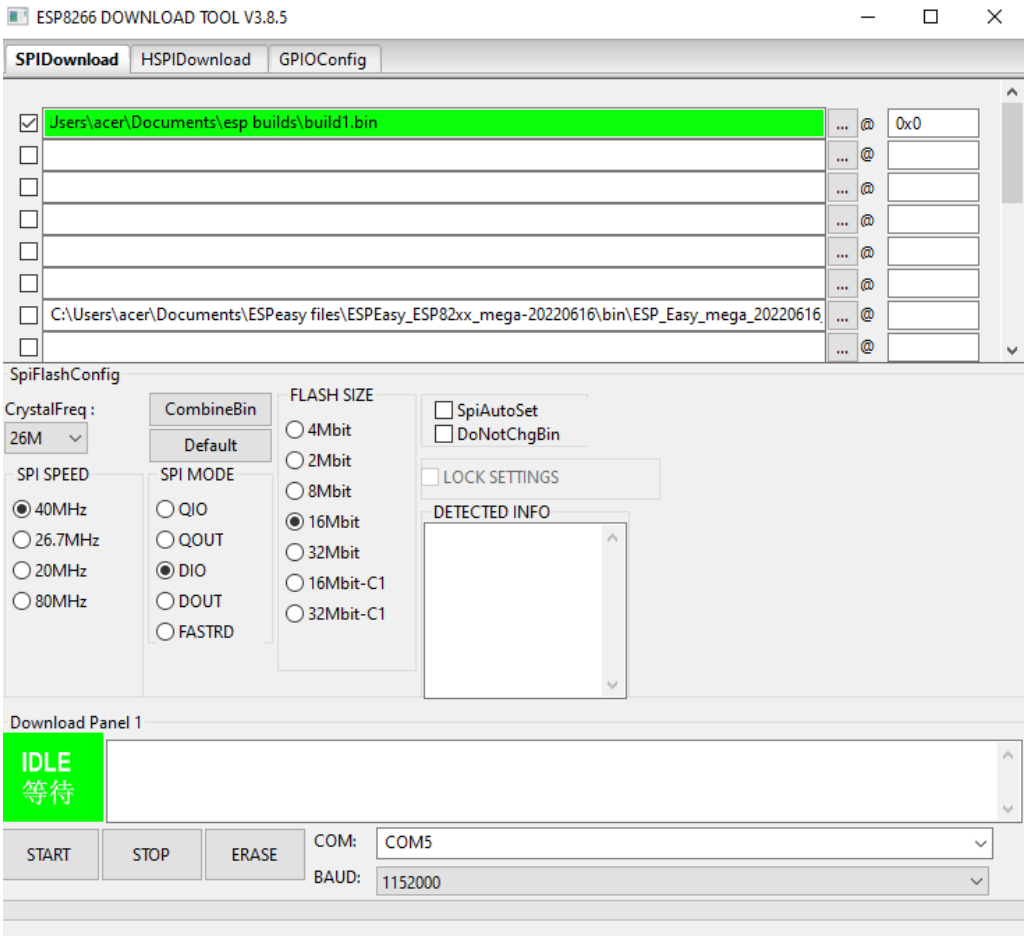
2. Select ESP8266 DownloadTool.



3. Select a build to flash on. It can be any build of esp8266 from anywhere on the computer you're using or from the bin directory of the ESPEasy_ESP82xx_mega-xxxx.
   Set the Address to 0x0 and tick the block on the left.



4. Select the flash size of the module. The esp8266 has 16Mb flash. Set the SPI mode to DIO. Also select the COM port the esp8266 is connected to and set the baud rate to 115200. (make sure the baud rate of the com port is set to 115200 in the device manager). Leave other settings as default and Select Start.

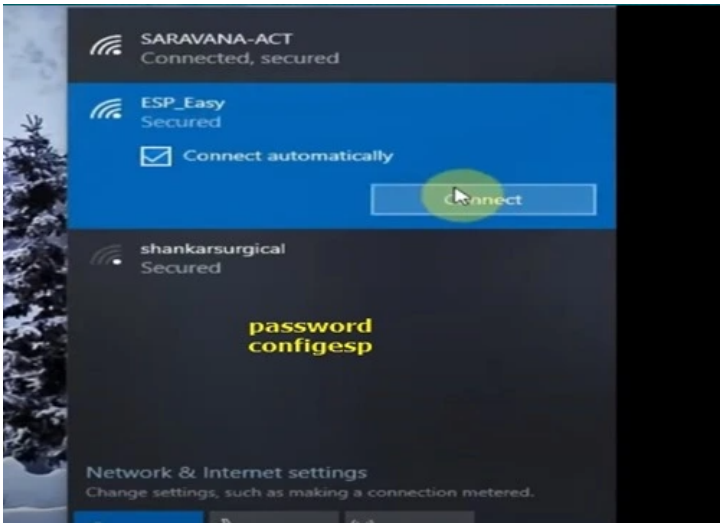


5. Once flashing to the board is complete, unplug the board then reconnect it to reboot it.

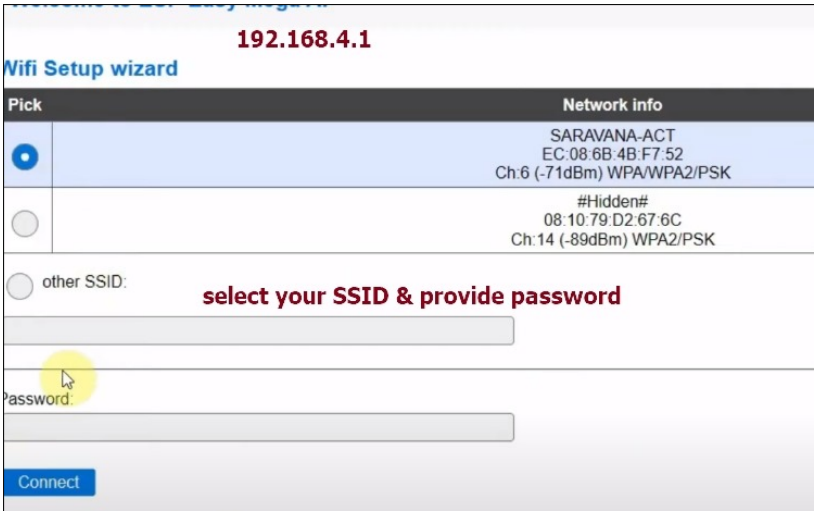
Connect to the device over wifi and connect it to a trusted LAN.

1. Once you've flashed a build of es8266 onto the board, the board should start a wifi access point named ESP-EASY. You eventually should see it as one of the Available networks on your computer (if wifi is enabled). If it does not show up after a while, reset the ESP board and wait again (It should take less than 2 minutes).

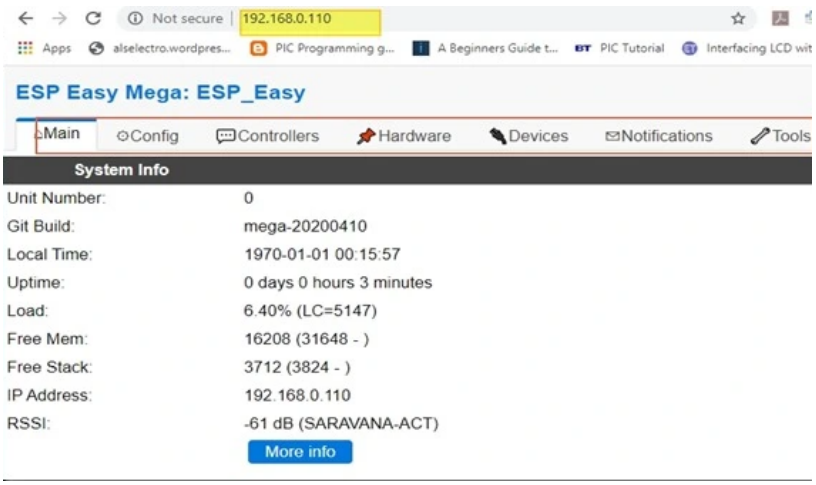
2. Connect to the access point. When prompted to enter a password, type in **configesp**. Once connected, on your preferred browser. Type in 192.168.4.1/setup, which will send you to a page to select which access point to connect to. It is advised to use the esp board in station mode, meaning you connect the board to a safe LAN your computer is connected to, and communicate with the board with the LAN.



3. Once you've selected which network to connect the esp board to, put in the password for the LAN if needed then connect. You'll be redirected to a page where you'll see a 20 second countdown of the device connecting to the network. Once the countdown is complete, you'll be redirected again to a page. This page contains the new ip address of the board in the LAN you connected it to. Disconnect your laptop from ESP-EASY and connect to the LAN.



4. In your browser, type in the new ip address of your board and press search. You will be directed to the main page of the ESPEasy build you flashed.



Appendix B: How to configure the OLED ssd1306 on ESPeasy

This tutorial online nicely describes how to use the OLED display with ESPEasy and the esp8266. https://emariete.com/en/connect-an-oled-display-ssd1306-to-espeasy/

Appendix C: How to use Rules

Rules are control instructions that act on any event generated by ESPEasy, such as when the system boots or during a gpio pin state change.
The link below gives a full description on how to enable and use Rules
https://espeasy.readthedocs.io/en/latest/Rules/Rules.html#

In summary, rules take the form

```
on ... do
  // Code in the event handling rules block
endon
```

Where between “on” and “do” you place the “Event” such as **System#Wake**. You can use the double forward slash to insert inline comments.

The example below sets a GPIO pin of the esp8266 on 5 seconds after the system is switched on.

on System#Wake do
timerSet,1,5 //Set timer 1 to count 5 seconds
endon

on Rules#Timer=1 do
GPIO,2,1 //when timer one is done counting, set the gpio pin to 1
endon

Appendix D: How to install and use PlatformIO on VS Code

Instruction on installing PlatformIO for VSCode can be found here:
https://espeasy.readthedocs.io/en/latest/Participate/PlatformIO.html#platformio-with-vs-code

Just make sure you use the latest version of Microsoft Visual Studio Code, the PlatformIO extension, git and Other extensions as well. I used PlatformIO on a laptop with 4GB RAM and a dual core intel processor and ran into a lot of compiling errors when building EASPEasy. I was able to successfully build using a laptop with an Intel i5 processor and 8GB RAM. Thus it seems PlatformIO is resource heavy when building ESPEasy, and not just any laptop can create a successful build.

Appendix E: Creating a simple plugin

A plugin can be thought of as an app for ESPEasy. It can be used to store variables, log data to another controller, or interact with peripherals, or whatever your imagination can create (given the system limitations). I could not find many intuitive online tutorials for creating plugins but the links below are a good start in understanding the firmware. I will give an overview of the main structure of a plugin given as much as I learned, in the form of examples.

NOTE: when writting this, I learned more about plugins but did not have access to the board to test a few features.

Useful links and documents to understanding the firmware and plugins.

1. The Plugin template file stored in **ESPEasy/srs/_Pxxx_PluginTemplate.ino** gives a typical skeleton of a plugin, but existing plugins can stray very far from this template.
2. https://www.letscontrolit.com/wiki/index.php/ESPEasyDevelopment briefly explains the data structures and code used by plugins. In the time of writing of this document it was still under development.
3. https://www.letscontrolit.com/wiki/index.php/ESPEasyDevelopmentGuidelines gives guidelines to adding plugins. Pay special attention to the naming conventions of plugins and the available ID's your custom plugins may use.
4. https://github.com/letscontrolit/ESPEasyPluginPlayground is where you find more guidelines on plugins and plugins under test.
5. https://www.letscontrolit.com/wiki/index.php/Official_plugin_list is the official plugin list.

Brief overview for creating a simple plugin.
Copy the contents of ESPEasy/srs/_Pxxx_PluginTemplate.ino, and in the same director of the file create a new file and paste what you've copied. Name the file _PXXX_<YourChoiveofName>.ino, I chose _P135_Blink.ino.

Giving a plugin a name

1. Plugins are referred to in the source code by their Plugin ID. Plugin ID's should be unique and numbered between 100 - 199 and filenames should use the "_PXXX_" prefix. Use the next available ID from the list of plugins. The maximum number of Plugins is defined in ESPEasy-Globals.h (PLUGIN_MAX). For example We can name this plugin _P135_Blink.ino, assuming 135 is the next available ID.
2. Where macros are defined (by the #define section) ensure you add the ID and name of the plugins to the correct macros as shown in the figure below

```
#define PLUGIN_135
#define PLUGIN_ID_135     135          // plugin id
#define PLUGIN_NAME_135   "Blink" // "Plugin Name" is what will be dislpayed in the selection list
```

3. In the files pre_custom_esp82xx.py and define_plugin_sets.h, add your plugin ID, exactly the same as other plugins are added.
4. Alternatively, you can copy an existing plugin such as Plugin 033. Create a new file and give it an appropriate name similar to above. Then, wherever there's a "P001" or

“001”, change it to “P135” and “135”, respectively. Then you can compile a build of ESPEasy and you should see a new plugin with the name you've given it appear at available devices.

includes

In the includes section, include these files. The first is necessary for all plugins created as it includes the necessary macros, data structures and more for creating a plugin.

```
// #include section
#include "_Plugin_Helper.h"
#include "src/Helpers/_Plugin_Helper_webform.h"
# include "src/ESPEasyCore/ESPEasyGPIO.h"
# include "src/Helpers/PortStatus.h"
```

Main structure of the plugin function

As seen in the plugin template, the most important function a plugin implements is the Plugin_xxx() function. Basically, most features of your plugin are configured in this function. The function makes use of a switch-case statement to control what the plugin should do by using what is passed in the **function** parameter. I will give examples of how to use some cases passed in **function**.

case function = PLUGIN_DEVICE_ADD:

This block is where you define the characteristics of your plugin. Here you use the DeviceVector struct called Device to define the characteristics of a plugin. The different values you pass to the attributes can be found in ESPEasy/srs/src/DataStructs/DeviceStructs.h.

```
case PLUGIN_DEVICE_ADD:
{
  // This case defines the device characteristics, edit appropriately

  Device[++deviceCount].Number           = PLUGIN_ID_135;
  Device[deviceCount].Type               = DEVICE_TYPE_SINGLE;
  Device[deviceCount].VType              = Sensor_VType::SENSOR_TYPE_SINGLE;
  Device[deviceCount].Ports              = 0;
  Device[deviceCount].ValueCount         = 3;
  Device[deviceCount].OutputDataType     = Output_Data_type_t::Simple;
  Device[deviceCount].PullUpOption       = false;
  Device[deviceCount].InverseLogicOption = true;
  Device[deviceCount].FormulaOption      = false;
  Device[deviceCount].Custom             = false;
  Device[deviceCount].SendDataOption     = false;
  Device[deviceCount].GlobalSyncOption   = true;
  Device[deviceCount].TimerOption        = true;
  Device[deviceCount].TimerOptional      = true;
  Device[deviceCount].DecimalsOnly       = false;
  break;
}
```

From the example above, “Number” sets the ID of the plugin. “Type” describes how the device is connected. The different configurations of type can be found in ESPEasy/srs/src/DataStructs/DeviceStructs.h from line 8 to 22. Here the plugin’s device is characterised by connection of one datapin.

“VType” describes the type of value the plugin will return, where i defined the plugin to return a single value.
“Ports” is used to select which port your device is going to use, I’ve selected none.
“ValueCount” describes the number of values the plugin will return, where in this case it will be three. “OutputDataType” describes a subset of selectable output data types which I defined as Simple.
“PullUPOption” allows us to set the internal pullup resistor to the data pin the plugin’s device is connected to, I set that as false.
“InverseLogicOption " allows you to use inverse logic at the data pin.
I was unsure what the “Custom” attribute does and the template does not give much detail about it. Naturally I left it as false.
“SendDataOption” allows the plugin to send data to a controller. I have not explored this option.
The “GlobalSyncOption” is no longer used, but I left it as true as other plugins leave it as true.
“TimerOption” allows you to set the Interval timer for your plugin, I have not explored what this does. I'm also uncertain what “TimerOptional” does.
“DecimalsOnly” allows you to set the number of decimals used in the returned values of your plugin. I set them to return no decimals.

case function = PLUGIN_GET_DEVICENAME:

This block for most plugins is important to get the plugin name. Most plugins use it as shown below.

```
case PLUGIN_GET_DEVICENAME:
{
  // return the device name
  string = F(PLUGIN_NAME_135);
  break;
}
```

case function = PLUGIN_GET_DEVICEVALUENAMES:

This allows you to add a new row for each output variable of the plugin. I chose "ValueCount" from the example above to be 3, thus each of the 3 values need a name. Outside the Plugin_xxx() function, in the #define section, you give the names of each value as shown below

```
#define PLUGIN_135
#define PLUGIN_ID_135     135          // plugin id
#define PLUGIN_NAME_135   "Blink" // "Plugin Name" is what will be dislpayed in the selection list
#define PLUGIN_VALUENAME1_135 "LEDState" // variable output of the plugin. The label is in quotation marks
#define PLUGIN_VALUENAME2_135 "DC_ON" // multiple outputs are supported
#define PLUGIN_VALUENAME3_135 "DC_OFF"
#define PLUGIN_135_DEBUG  false         // set to true for extra log info in the debug
```

Then in the PLUGIN_GET_DEVICEVALUENAMES block, allow the names of the values to be displayed in the module configuration page

```
case PLUGIN_GET_DEVICEVALUENAMES:
{
  // called when the user opens the module configuration page
  // it allows to add a new row for each output variable of the plugin
  // For plugins able to choose output types, see P026_Sysinfo.ino.
  strcpy_P(ExtraTaskSettings.TaskDeviceValueNames[0], PSTR(PLUGIN_VALUENAME1_135));
  strcpy_P(ExtraTaskSettings.TaskDeviceValueNames[1], PSTR(PLUGIN_VALUENAME2_135));
  strcpy_P(ExtraTaskSettings.TaskDeviceValueNames[2], PSTR(PLUGIN_VALUENAME3_135));
  break;
}
```

case function = PLUGIN_SET_DEFAULTS

This is used to set default configurations. In my example i left this empty.

```
case PLUGIN_SET_DEFAULTS:
{
  // Set a default config here, which will be called when a plugin is assigned to a task.
  success = true;
  break;
}
```

case function = PLUGIN_WEBFORM_LOAD:

This block allows you to add html forms on the plugin configuration page.
In my example, I added one form that allows a person to set a value called p135_dimvalue. This value can be accessed by the plugin in other blocks of code.

```
case PLUGIN_WEBFORM_LOAD:
{
  // this case defines what should be displayed on the web form, when this plugin is selected
  // The user's selection will be stored in
  // PCONFIG(x) (custom configuration)
  // after the form has been loaded, set success and break
  addFormNumericBox(F("On Duty Cycle value"), F("p135_dimvalue"), PCONFIG(0), 0, 100);


  success = true;
  break;
}
```

case function = PLUGIN_WEBFORM_SAVE:

Thai block is used whenever a form is submitted. I haven't used this yet but it is useful for saving values in a form that was filled.

```
case PLUGIN_WEBFORM_SAVE:
{
  // this case defines the code to be executed when the form is submitted
  // the plugin settings should be saved to PCONFIG(x)
  // ping configuration should be read from CONFIG_PIN1 and stored

  // after the form has been saved successfuly, set success and break
  success = true;
  break;
}
```

case function = PLUGIN_INIT:

This block is used whenever a plugin should be initialised. For example, you can configure GPIO Pins as input or output here.

```
case PLUGIN_INIT:
{
  // this case defines code to be executed when the plugin is initialised

   ///Create var to store LED status
  portStatusStruct newStatus;
  const uint32_t   key = createKey(PLUGIN_ID_135, CONFIG_PIN1);
  newStatus = globalMapPortStatus[key];
  newStatus.state                                          = P135_LED_OFF;
  newStatus.output                                         = newStatus.state;

  //set GPIO as output
  pinMode(CONFIG_PIN1, OUTPUT);//will be used for digital write for Duty Cycle On
  newStatus.mode = PIN_MODE_PWM;
  savePortStatus(key, newStatus);

  // after the plugin has been initialised successfuly, set success and break
  success = true;
  break;
}
```

I haven't tested the code above enough, but it should give clues as to how to configure pins.

case function = PLUGIN_EXIT:

This block is used for cleanup tasks. I haven't tested an example of this, but you can see examples of how it's used in other plugins.

```
case PLUGIN_EXIT:
{
  // perform cleanup tasks here. For example, free memory

  break;
}
```

case function = PLUGIN_ONCE_A_SECOND or _TEN_PER_SECOND:

These blocks, as described below, can be used to execute code once per second like polling a serial device or toggling a gpio pin. I haven't got a screenshot of a working example of how to use this.

```
case PLUGIN_ONCE_A_SECOND:
{
  // code to be executed once a second. Tasks which do not require fast response can be added here

  success = true;
}

case PLUGIN_TEN_PER_SECOND:
{
  // code to be executed 10 times per second. Tasks which require fast response can be added here
  // be careful on what is added here. Heavy processing will result in slowing the module down!

  success = true;
}
```

Appendix F: VE.Direct Protocol

Use the link below to understand the VE.direct protocol and devices. Page 4 gives detail on which form data is transferred.
http://www.atakale.com.tr/image/catalog/urunler/charger/victron/pdf/victron_energy_haberlesme_protokolu_VE.Direct-Protocol-3.29.pdf

The link below takes you to a github page containing a library useful for communicating with most VE.direct devices. Note the serial port for VE.direct the esp8266 board uses is the same serial port to flash via usb. Given the data sent is asynchronous, you can log debug information via the OLED or through the wifi network connection.

https://github.com/winginitau/VictronVEDirectArduino

The link below contains a library for communicating with the OLED display if you decide to communicate with it without ESPEasy

https://github.com/adafruit/Adafruit_SSD1306