

EVTV Battery Monitor Display



for the
ESP32 Tesla Battery Module Controller

INTRODUCTION

EVTV has developed several devices to monitor Solar Energy Storage based on the Tesla Model S battery module.

This document describes an optional 7-inch touchscreen display that presents information from the EVTV ESP32 Battery Module Controller to allow visual presentation of battery operating parameters in real time.

The display is based on the popular Raspberry Pi credit card sized single board computer model 3B+ released in 2018.

The decline in numbers and skills of students applying for Computer Science at the University of Cambridge Computer Laboratory led to the formation of a team headed by Eban Upton including Rob Mullins, Jack Lang, and Alan Myscroft. Beginning in 2006 they envisioned a very small and very affordable computer students could use as an introduction to computer science.

In 2011 they introduced the Raspberry Pi. It was originally designed to display on a television and use any USB keyboard for input. And it had a full Linux kernel and implementation on the system. They did their own Linux distribution, termed Raspian, based on the popular Debian distribution.

In the past seven years over 20 million of these tiny computers on a credit card have been sold at a very inexpensive \$35.

The Model 3B+ was released in 2018 and really achieves a remarkable level of utility providing a reasonably complete Linux operating system.

Broadcom BCM2837B0 Quad Core 1.4 MHz 64-bit ARM Cortex 53 processor.

1 GB of RAM

4 USB ports – Microchip LAN7515.

1 Gigabit Ethernet Port.

1 HDMI video port.

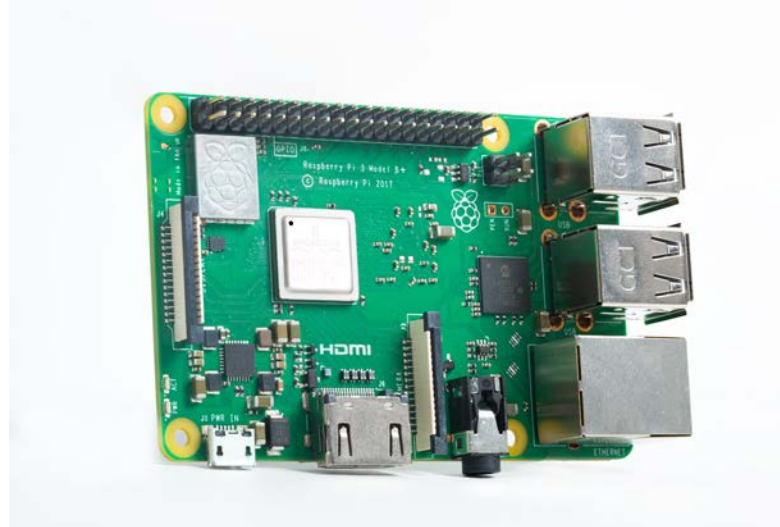
1 camera port.

General Purpose Input/Output (GPIO) port.

Built-in Cypress CYW43455 WiFi radio on both 2.4GHz and 5 GHz bands.

Built-in Bluetooth BLE radio.

32GB MicroSD card memory



This single board computer provides us a full Linux Raspian environment with the GNU GCC and G++ compilers.

The Raspberry Pi Foundation has also provided a very attractive 7-inch touch screen display.

We have combined that with a sturdy enclosure containing both the Raspberry Pi computer, and the touchscreen, and a very small wireless keyboard necessary to set a few configuration items on the system.



We have configured this system so that the EVT V Battery Monitor Display software automatically loads on powerup.

Further, the system has the RealVNC host screen sharing software already installed and configured to allow you to connect to the device using any VNC client equipped laptop to view and control the device remotely. With proper IP routing, you could conceivably access this system from anywhere.

In its first iteration, the EVTV Battery Display uses User Datagram Protocol (UDP) packets to communicate with the ESP32 Battery Module Controller using the broadcast IP address. And so if they are both connected to the same TCP/IP Access Point (your home wireless router) the EVTV Battery Display will present data from you Battery Module Controller.

In this way, you can put the display anywhere in your home to monitor your solar energy storage system with no wired connection at all.

Additionally, the device has the ability to pass data using Amazon’s AWS Iot Core cloud service to echo data to a second display located anywhere worldwide.

And finally the display itself can act as a web server to display historical data held in an internal MySQL database.



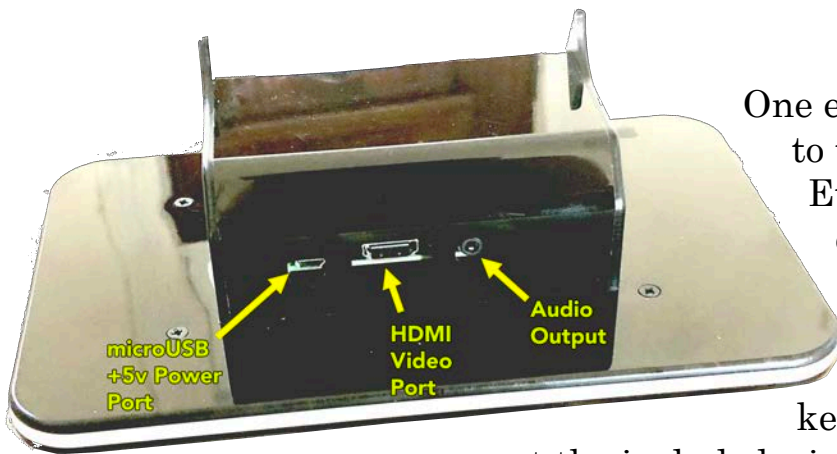
INCLUDED

EVTV Battery Monitor Display. The display comes assembled with the Raspberry Pi 3B+, a 7-inch capacitive touch screen 800x480 display, and enclosure.



The enclosure provides access to several ports on the Raspberry Pi 3B+ unit.

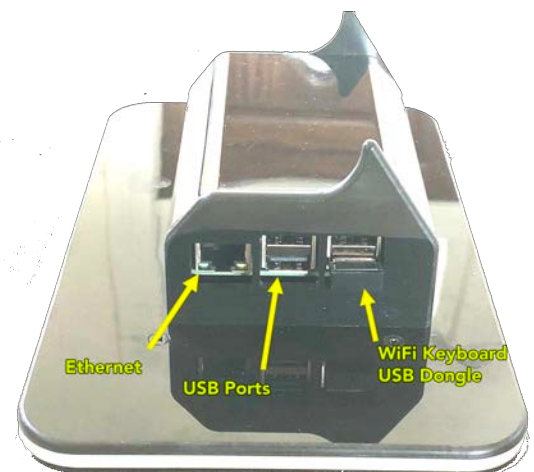
On the bottom of the unit are three ports. A standard audio output port, an HDMI video port. And a microUSB port. The microUSB provides a handy place to connect the +5v USB power adapter. HDMI and audio ports are not normally used for this application.



One end of the device provides access to the communication ports. The Ethernet port allows direct wired connection to your Access Point if desired.

One USB port hosts the keyboard WiFi dongle used to connect the included wireless mini-keyboard.

The remaining 3 USB ports are available.



We have included a small wireless keyboard. This allows entry of a couple of configuration items and selection of a WiFi Access Point from the Raspbian Desktop.



The keyboard is battery powered. It comes with its own USB cable but this is not necessary to connect to the display. It is provided only to recharge the keyboard battery.

The keyboard also comes with a WiFi dongle. This must be installed in one of the display USB ports. Configuration of the display for the keyboard is automatic.



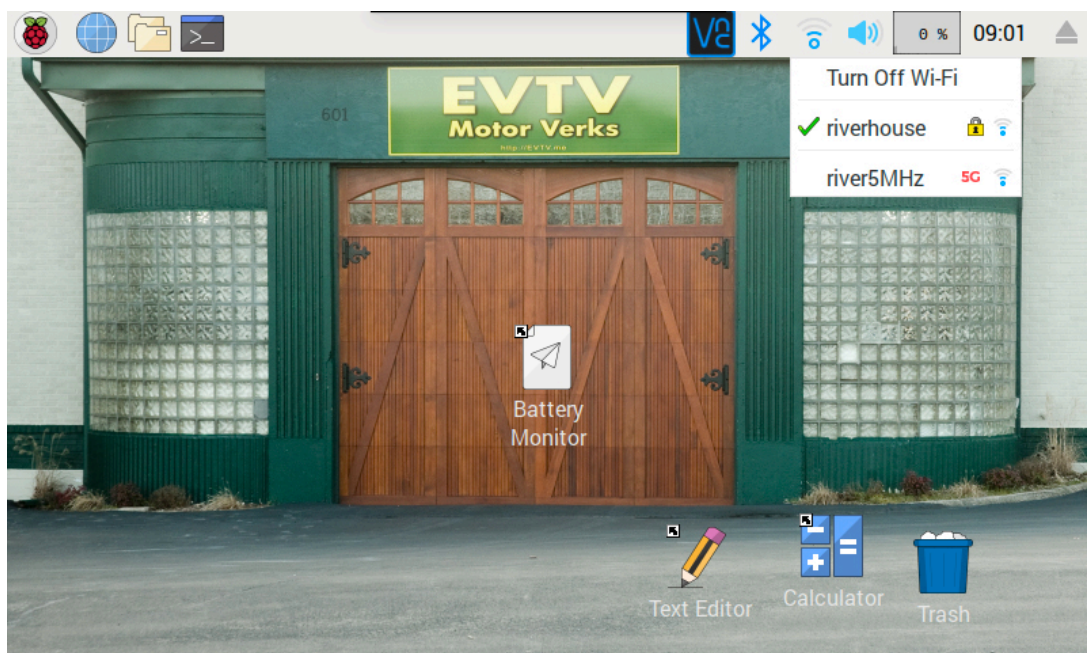
Finally, we provide a +5v microUSB power adapter. Plug this into any 120vac house outlet and connect to the microUSB port on the display to power up.

SETUP AND CONFIGURATION

On power up, the system will automatically load the Battery Monitor display program. Predictably enough, the first thing we want you to do is escape it.

Press the GEAR symbol in the upper right hand corner of the display to call up the configuration screens. Screen 3 has an EXIT button that will allow you to escape to the Raspian Desktop.

In the upper right hand corner on the tool bar select the WIFI symbol. This will call up a menu of available WiFi Access Points. You may have to wait for a minute or two for this list to fully populate.



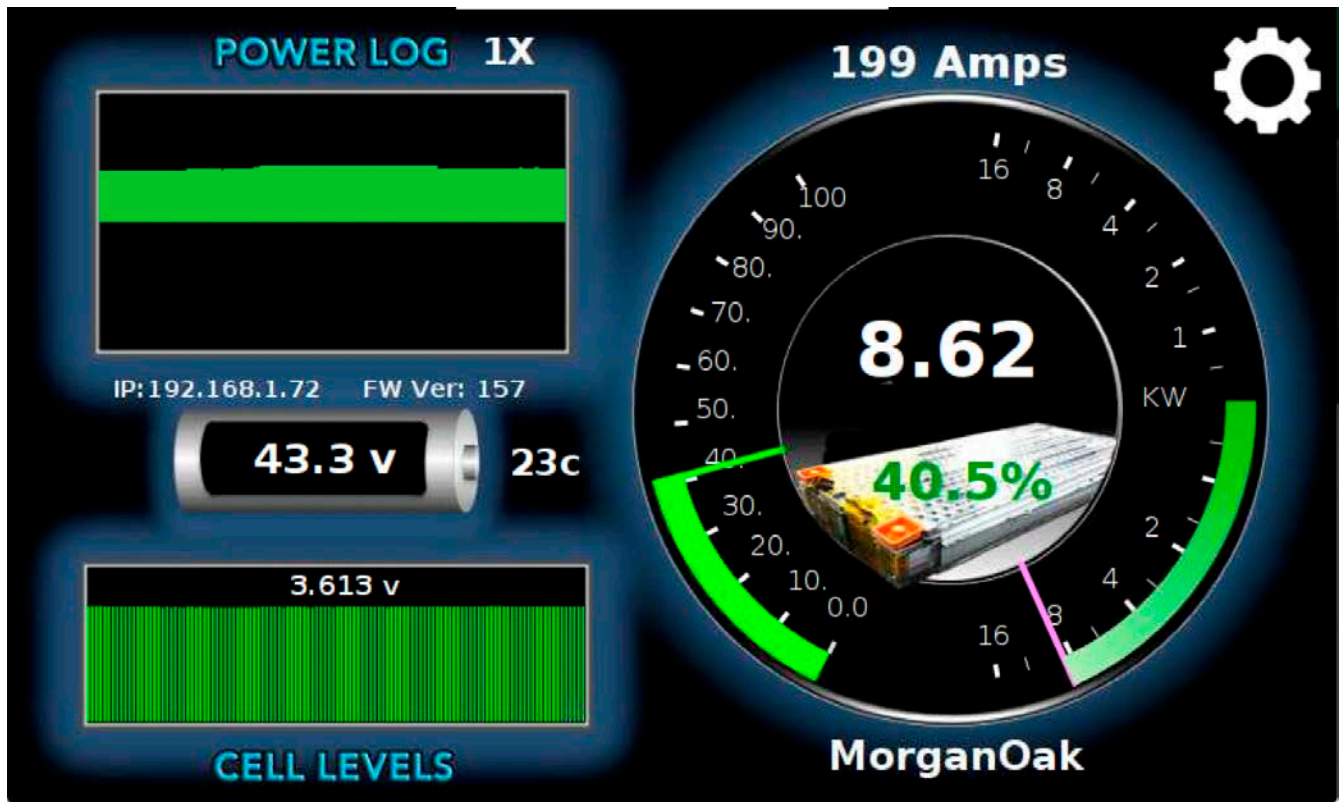
Select your AP from the list and enter your password. This should then indicate a connection to your AP.

Once connected, simply click on the Battery Monitor program in the center of the desktop to bring up the Battery Display program. Or power cycle the display by removing and reinserting the power adapter to reboot the

program. The unit should boot up into the Battery Monitor Main display screen once more and you are in operation.

THE MAIN DISPLAY SCREEN

The main display screen provides access to a wealth of information about battery operation, as well as access to other subfunctions of the device.



The right half of the screen features a large display dial fashioned after the Tesla Model S Speedometer

The left side of this dial displays the current average cell voltage for the entire pack. A lower white tick indicates your LOVOLT cutoff setting and a similar tick on the upper part shows the HIVOLT cutoff setting for your battery.

Arc of blue shows your current average cell voltage and moves up and down with charging and discharging.

The right half of this screen provides the current power output of your battery pack in kiloWatts. The lower half, like the Tesla display, shows regenerative braking, although in this case more likely solar charging coming IN to your pack as a positive current.

The upper half displays in red and indicates a negative current and power output from the battery to your power inverter.

The center of the dial provides a digital indication of this power level again in kiloWatts.

Beneath that is an image of an actual Tesla Battery Module with a percentage value representing current state of charge SOC.

Just above the dial is a digital indication of current amperes with positive values indicating charging and negative values indicating discharge.

Immediately to the left of the dial is a small battery depiction providing the total voltage of your battery pack as configured and a temperature indication representing the HIGHEST temperature of ALL terminal temperatures monitored.

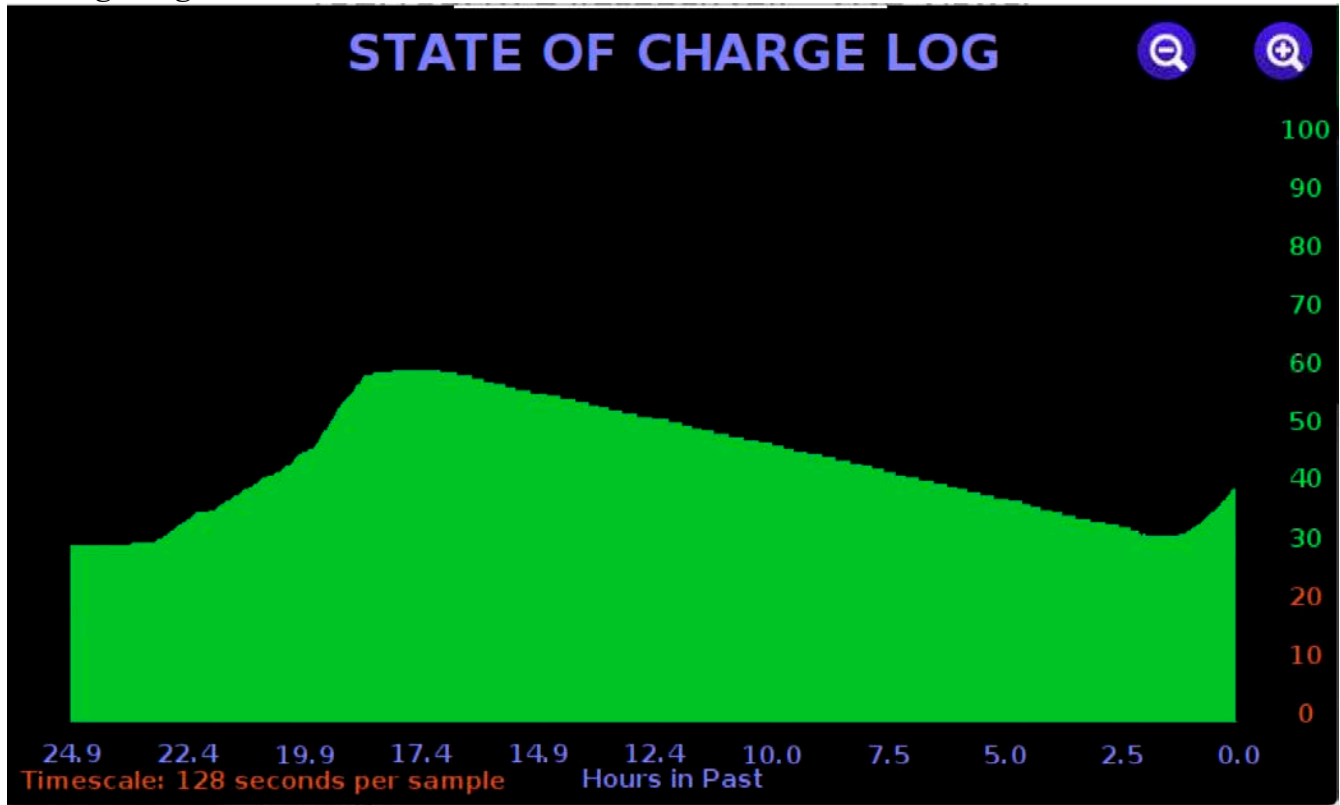
Beneath the small battery is a **CELL LEVELS** window graphing the voltage of each individual cell in your pack. The average cell voltage is also provided digitally.

Above the small battery is the **POWERLOG** window. This logs inflows and outflows of power relative to the battery over time. Note the small white 1X above it. Pressing this changes the scale of the power logging window to 2X. This allows you to magnify the display to show small power levels more readily.

Finally, in the upper right hand corner of the display is a small white GEAR symbol. Press this to access the configuration screens.

SOC LOG DISPLAY

On the main power dial, press the digital SOC value to access the State of Charge log screen.



This screen allows you to view a log of State of Charge values over time. The left and right arrows allow you to increase or decrease the scale of time displayed. The right arrow decreases log time to a minimum value of 1 second per vertical bar. This corresponds to a log of the last 11.7 minutes.

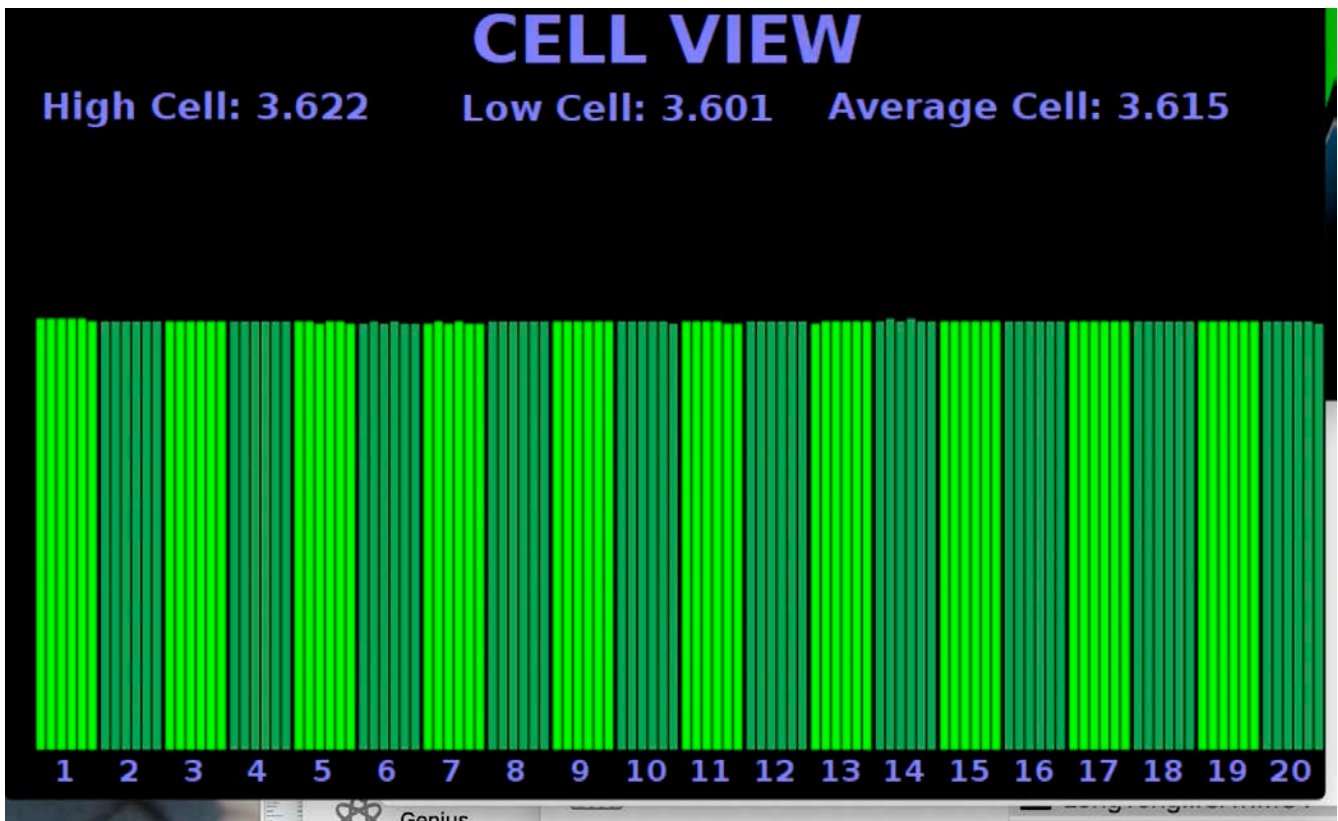
The left arrow increases the span in time in a series of doubling increments to a maximum value of 512 seconds. In this display, each vertical bar represents a sample SOC taken each 512 seconds and spanning the past four days.

Note that this is a display function. Actual logging is performed each second in any event.

Press anywhere in the center of the display to return to the main screen.

INDIVIDUAL CELL STATUS

Pressing the **CELL LEVELS** display in the lower left portion of the Main Display calls up a more detailed cell status display.



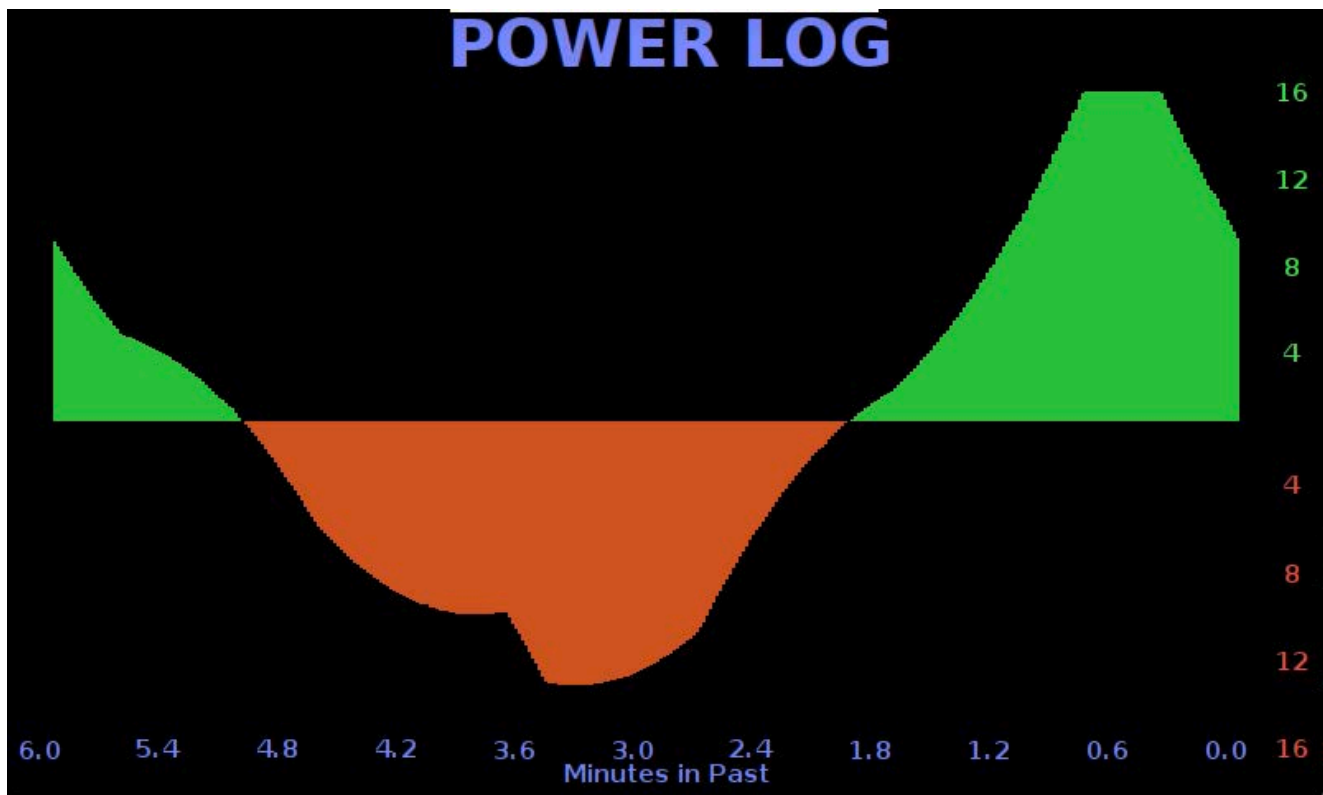
This display graphs the voltage of each of the cells in your pack up to a maximum of 40 cells. These vertical bars are arranged in modules of six cells and each value is updated every second or so with data from the ESP32 Battery Module Controller.

It also provides the voltage of the highest cell, the lowest cell, and the cell average voltage digitally.

The number and width of cell bars displayed is a function of the number of cells discovered and tracked by the battery controller.

POWER LOG

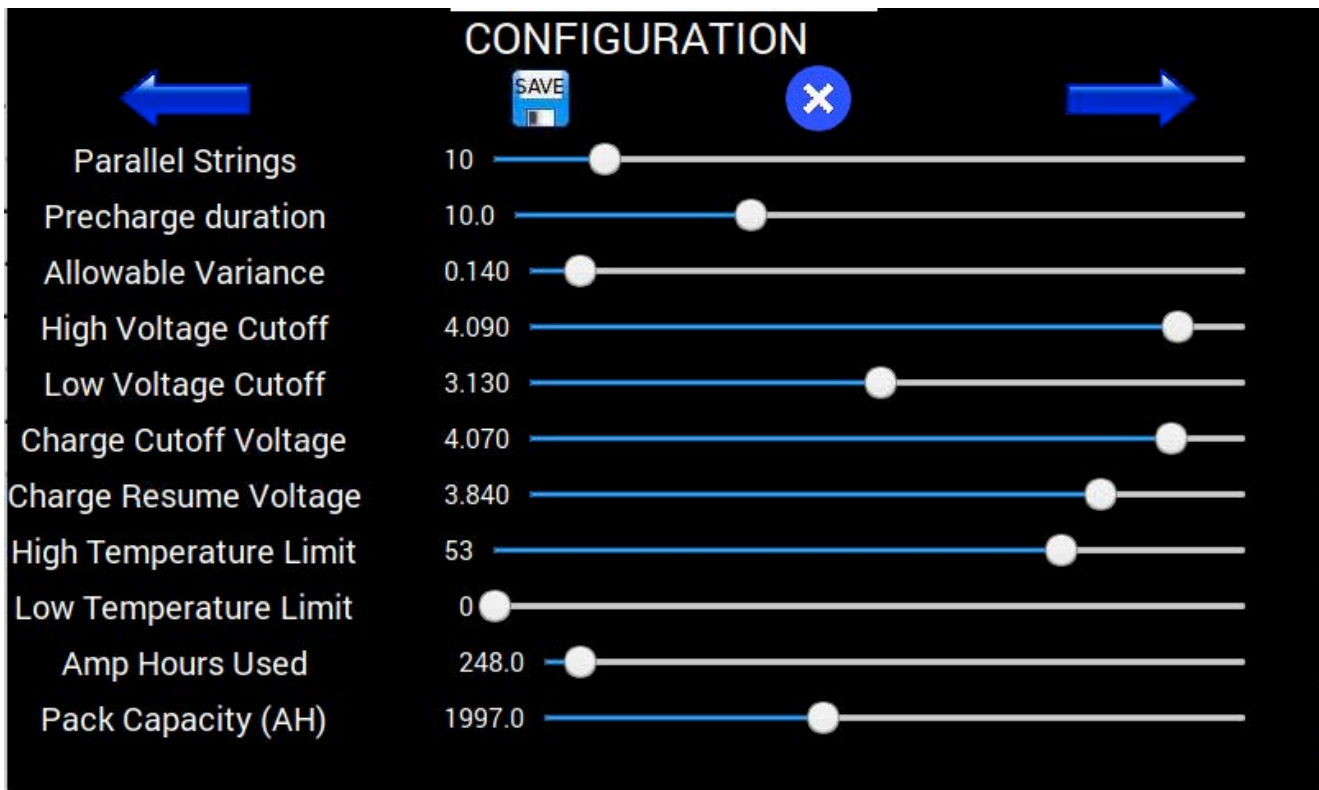
Pressing the Power Log display in the upper left corner of the main panel calls up the Power Log Display. This is an expanded and enlarged version of the POWER LOG panel.



This is useful in that it depicts the power level and allows you to observe variations in power input and output caused by passing clouds and increasing loads as various appliances come on and turn off.

CONFIGURATION SCREENS

Press the small white gear in the upper right hand corner of the main panel to access a series of configuration screens.



The battery controller has a number of configuration items important to the operation of the battery. These are normally set using a very plain ASCII text display and a somewhat complicated series of input string labels and values entered from the keyboard.

The controller sends this configuration data along with the usual operation data via User Datagram Protocol packets. In this way, neither the display or controller needs to know the IP address of the other.

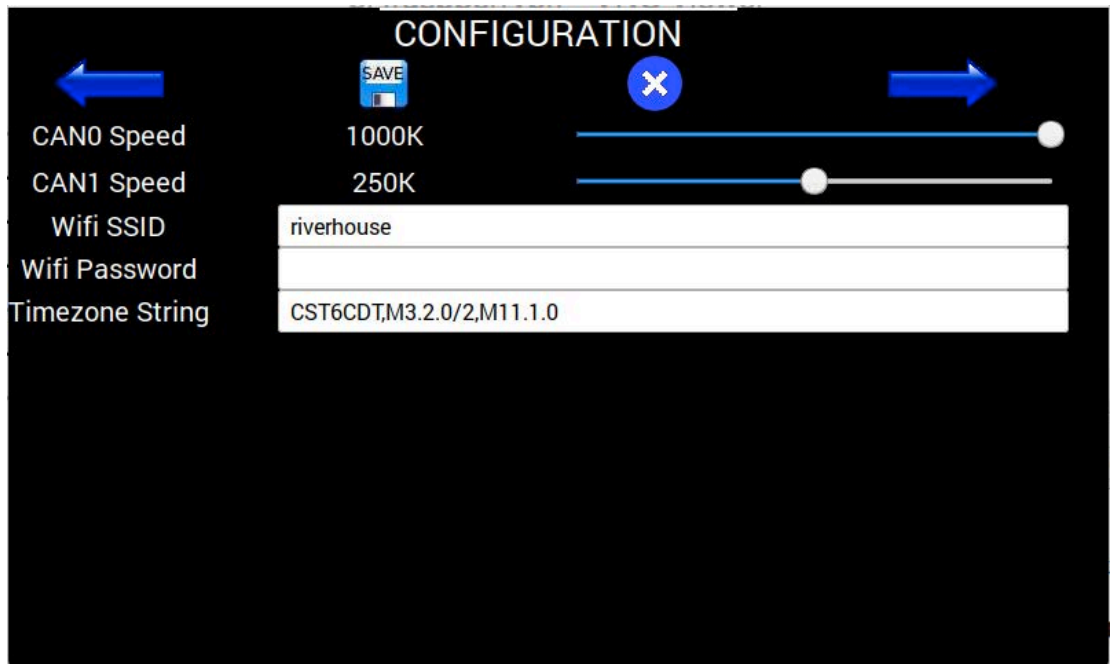
This first screen provides information on your current configuration. But it also allows you to modify these values using convenient sliders. If you change any of these values you must press the SAVE icon to actually

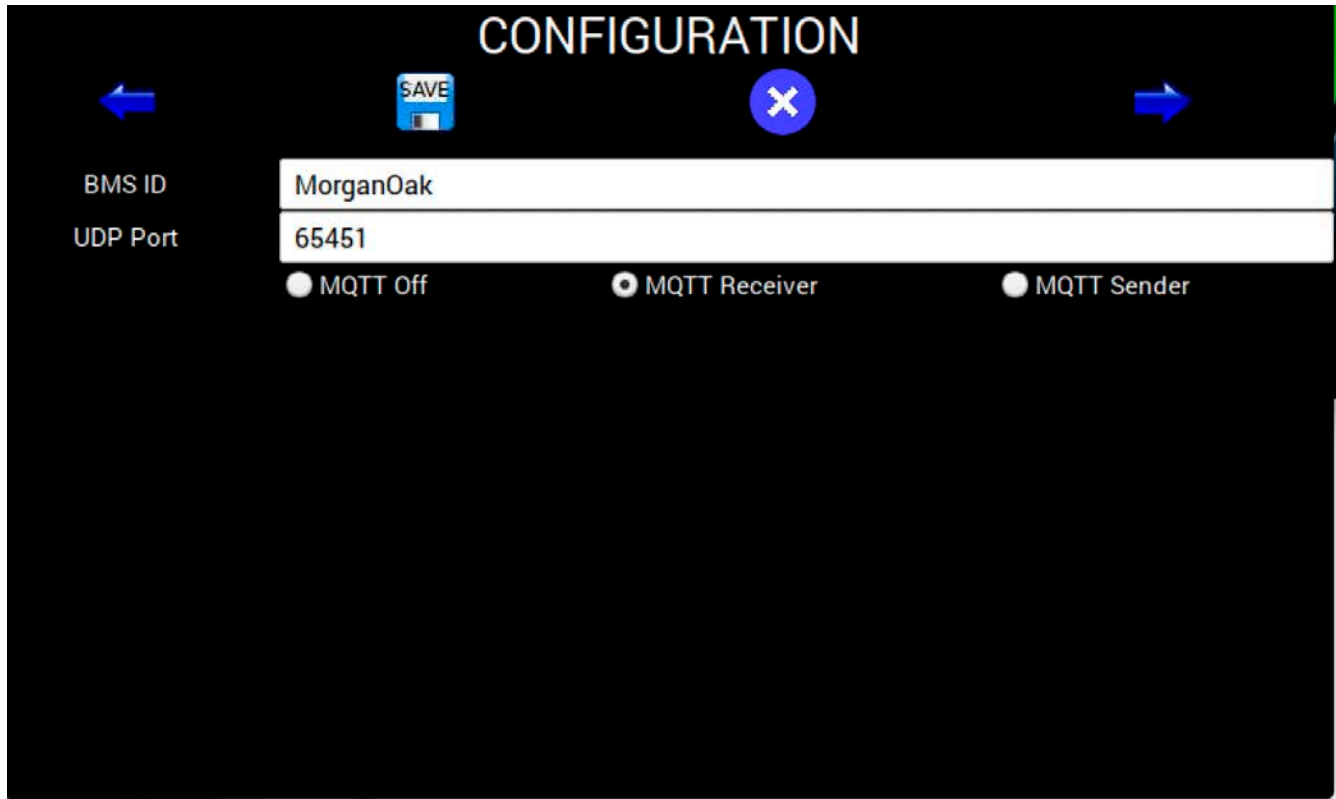
transmit them to the controller. They will have no effect or persistence until you press SAVE.

When you do,, the display can detect the IP number of the controller from the UDP packets. It then uses that IP number to connect via the Transmission Control Protocol, which provides a greater level of error checking and security than UDP. Configuration data is then transmitted to the controller via TCP.

The X icon allows you to escape back to the main display screen.

Left and right arrow icons allow you to access the other config screens.





BATTERY LOG FILE

The Battery Display provides logging windows for both Power and State of Charge and these provide a powerful glimpse into the operation of your battery. But beyond that, they provide a visual grasp of your entire solar power system, allowing you to monitor power IN to the system from photovoltaic arrays and power OUT of the system through your inverter.

Ultimately the battery was ALWAYS destined to be the heart of a home power system and the Battery Display makes this all too evident.

By monitoring the SOC LOG over several days, you can easily see how high your battery charges during the day, and low it goes while providing power at night. This is important to correctly sizing the balance between battery capacity and photovoltaics to your application. Your loads vary with day and

season as does solar irradiance. Few realize HOW MUCH this varies. You can expect 3.5x power in June compared to December in the north hemisphere.

The POWER LOG allows you to view the past few hours showing every passing cloud as charging IN to the battery varies during the day and the impact of every appliance coming on day or night. It all shows up in the Powerlog.

But realistically there is only so much we can present on a 7-inch touch screen. And particularly for those experimenting and innovating with their solar system, more detailed analysis may be desired.

Underlying these simple displays is actually a data logger that logs a few key data elements received from the ESP32 Battery Module Controller. Data log entries are time stamped and logged every second to a comma delimited text file titled **PowerSafe100.log** that makes it easy to import days, weeks or months of data into a Microsoft Excel Spreadsheet for example.

This seems a bit wasteful of storage space and CPU time but it is actually quite important. The EVTV Battery Monitor features an unusually large 32GB SD card for this reason alone.

```
08/27/2018 19:19:41 ,11.5,41.1,-34,-1397,-1415.70,036
08/27/2018 19:19:43 ,11.5,41.1,-34,-1397,-1415.70,036
08/27/2018 19:19:45 ,11.5,41.1,-34,-1397,-1415.70,036
08/27/2018 19:19:47 ,11.5,41.1,-34,-1397,-1415.60,036
08/27/2018 19:19:49 ,11.5,41.1,-32,-1315,-1415.60,036
08/27/2018 19:19:51 ,11.5,41.1,-30,-1233,-1415.60,036
08/27/2018 19:19:53 ,11.5,41.1,-33,-1356,-1415.60,036
08/27/2018 19:19:55 ,11.5,41.1,-34,-1397,-1415.60,036
08/27/2018 19:19:57 ,11.5,41.1,-34,-1397,-1415.50,036
08/27/2018 19:19:59 ,11.5,41.1,-34,-1397,-1415.50,036
08/27/2018 19:20:01 ,11.5,41.1,-34,-1397,-1415.50,036
08/27/2018 19:20:03 ,11.5,41.1,-34,-1397,-1415.50,036
```

Here we see just a snippet of a **PowerSafe100.log** file. Each entry is less than 60 bytes, and after the operating system and program there are probably 26GB of space remaining on the SD card. And so in theory we can

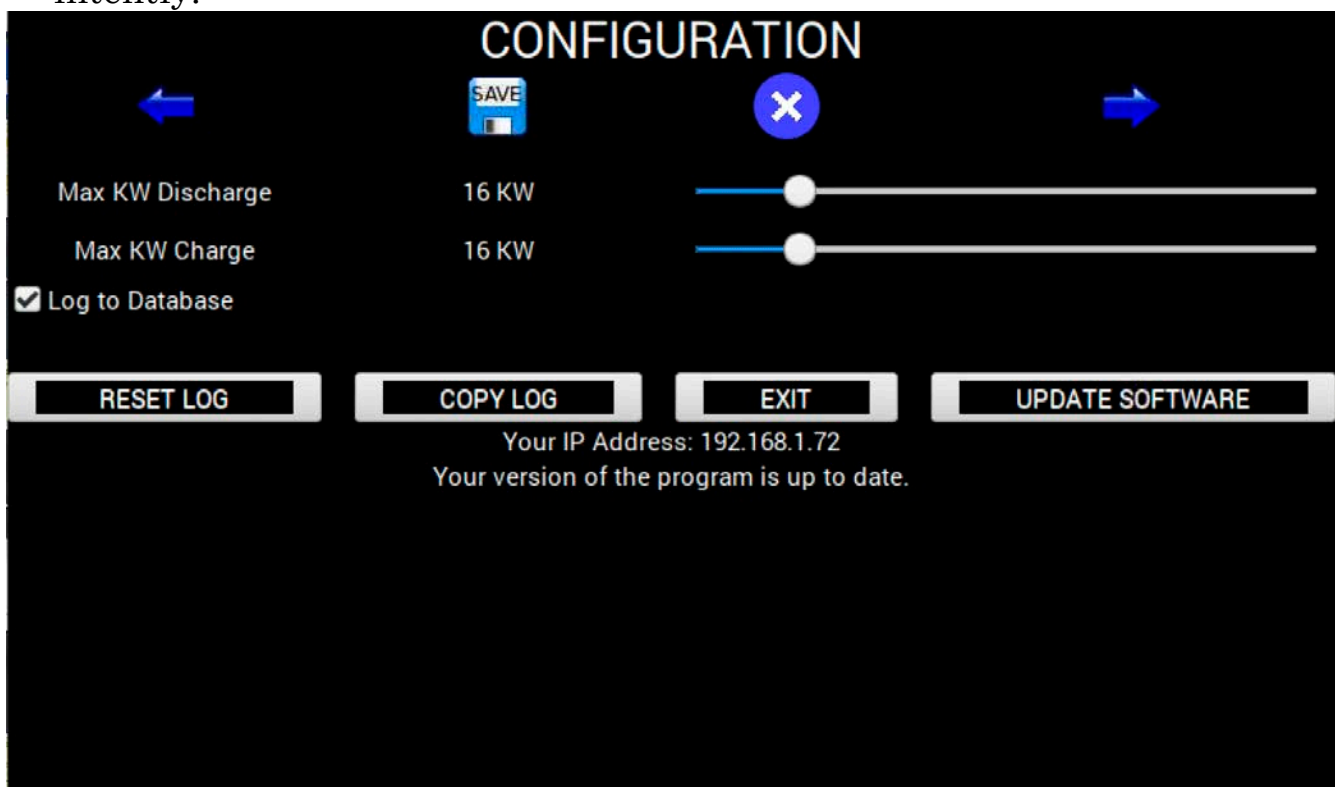
log about 824 years of solar data in this format recording a sample each second.

Each entry is time stamped with the data and time. The data follows in comma-separated columns. From left to right:

COLUMN	DATA
1	State of Charge
2	Voltage
3	Current in Amperes
4	Power in watts
5	AH used
6	Temperature of highest terminal measured.

And so you can see this comma separated format could be easily imported into a more advanced database or spreadsheet program for more detailed analysis.

But the data is held on the 32GB microSD card inside the enclosure housing the Raspberry Pi. We caution you NOT to remove and reinstall this card as it is actually quite fragile and holds the operating system and software to run the device. We've broken several of these just by looking at them really intently.



Fortunately, the Raspberry Pi 3B+ features four USB ports accessible in the enclosure. If you insert a FAT formatted USB thumb drive into one of the USB ports you can easily copy PowerSafe100.log to the drive. The third configuration screen is shown above. It has several handy buttons.

The **EXIT** button allows you to quit the program to the Pi Desktop to set up your WiFi for example. A single program is centered on the Desktop titled Battery Monitor. Click this to bring the battery monitor display program again.

The **RESET LOG** button will erase the current **PowerSafe100.log** file and exit the program. When you bring the program back up, it will start with a fresh and empty PowerSafe100.log file. Note that ALL data in the previous log will be lost.

The **COPY LOG** button is very handy. Clicking this causes the current **PowerSafe100.log** file to be COPIED to a USB thumb drive.

There is one proviso. The thumb drive can be any size large enough to hold the file, but it **MUST BE NAMED “BATTERY”** and best if formatted for MS-DOS FAT. You can easily change the name of a thumb drive on any PC. But we have also included a 16GB drive with your display and appropriately named **BATTERY**.

UPDATE SOFTWARE button. The EVTV Battery Display software is guaranteed to be 100% perfect in all respects. But as United States Congressman Thomas Massey of Kentucky, who inspired this project, is wont to say “We respectfully reserve the right to revise and extend our comments into the public record.”

It is POSSIBLE that in the future, we might find issues with the Battery Display or find advantage in additional features. For this reason, we refer to the operating software as **SOFTware**.

In the event that your Battery Display is connected to a WiFi access point and IF that AP has a further connection to the global Internet, the **UPDATE SOFTWARE** button can be useful. If you touch this button, the program

downloads the latest current version of the program from Amazon's AWS cloud service, installs it, and restarts the device.

This may take a few seconds, but when it restarts it will run the new revised and extended comment version.

With no connection to the Internet, this button will simply reboot the device, but no new version will be installed.

COMMUNICATION INTERFACE

The EVTV Battery Display was designed specifically for use with the ESP32 version of the EVTV Battery Module Controller which in turn is specific to providing an interface to the Tesla Model S Battery Modules.

And that is the only use supported by EVTV formally.

That said, we are here providing the communications interface for the device. Batteries are batteries are batteries. And all battery management applications have most of the same features in common. ALL batteries have a state of charge, an input or output current level, a voltage level, power levels, and most provide access to individual cell voltage levels.

And so in theory, this display could be interfaced to almost any battery management system.

Essentially, the EVTV Battery Display uses the Controller Area Network (CAN) protocol 2.0A with 11 bit message identifiers. We originally configured it to use a third party CAN hat for the Raspberry Pi to receive CAN frames through the usual twisted pair CAN wired interface and indeed it can still be configured to do just that and it will work identically in all respects.

The development of the UDP WiFi interface was driven by a desire to separate the display physically from the physical battery assembly. In this way, you could have the display in the kitchen for example, while the battery itself was in an outbuilding or garage in a solar storage application. Similarly you could dash mount the display and have the battery itself elsewhere on a car or boat with no need to run hard wiring between them.

If using hardwired CAN bus connections, any device can produce the necessary CAN message and it will be received and displayed by the program. The configuration screens make no real sense or have application in other environments, but the displays can all be used quite successfully.

UDP COMMUNICATIONS

In the case of the WiFi interface, things get a little less standard and a wee bit more complicated. The Battery Display monitors for incoming UDP packets on port 6500. Any it receives will be interpreted as CAN frames.

The problem of course is there is no standard for UDP encapsulation of CAN.

In the case of WiFi interface, we simply encapsulated the raw CAN data structure we use in our own CAN libraries for the ESP32.

These are based on Collin Kidder's `esp32_can` library.

<http://github.com/Collin80>.

Much of this has no application of interest because it relies on our CAN libraries and we simply shuttle the entire structure peculiar to our libraries back and forth over UDP. This makes programming UDP, for us, just like programming CAN.

The basic EVTV CAN data structure is 24 bytes:

- BYTE 0-7** **The 8 data bytes.**
- BYTE 8-11** **uint32_t Message ID 29 bit if IDEextended set, 11 bit otherwise**
- BYTE 12-15** **uint32_t family ID - used internally to library**
- BYTE 16-19** **uint32_t timestamp recording when mailbox message was received.**
- BYTE 20** **uint8_t RTR - Remote Transmission Request (1 = RTR, 0 = data frame)**
- BYTE 21** **uint8_t priority - only important for TX frames and not often then**
- BYTE 22** **uint8_t Identifier Extended flag 1 eq 29-bit 0 for 11 bit**
- BYTE 23** **uint8_t length - number of data bytes**

And so any UDP transmission with a raw data packet of 24 bytes on port 6500 will be received and interpreted if it is a CAN message ID of interest.

The ESP32 controller actually broadcasts these packets on the IP broadcast address (.255) on port 6500. In this way they go to ALL devices connected to the Access Point and so the controller doesn't need to know the IP number of the Battery Display or even if a Battery Display exists. The user can simply configure the Battery Controller to send UDP or not.

And so the Battery Display simply monitors its' own IP number for ANYTHING coming in on port **6500**. If a message ID of interest appears in bytes 8-11 of the packet, it processes the packet.

So don't be daunted by the structure. You don't need to put ANYTHING in most of the bytes in it. The data goes in the first eight bytes. The message ID, always 11bit, follows in Bytes 8-11, and the number of data bytes actually used goes in the last byte 23. Set all the others to zero. They have to be there. But the Battery Display doesn't actually process them.

And so, you can display anything you want on the Battery Display by connecting to the AP and sending 24 byte UDP packets to port 6500 on the broadcast IP address (**.255**).

CAN MESSAGE IDS

Whether transmitted by actual CAN differential wire pair or by UDP packet, the CAN message definitions are identical. You will note some odd usage here and unexplained gaps in the bytes used.

We used a display interface from Andromeda Interfaces termed the EVIC. We developed software to use this with our Tesla Model S Drive units. When we started employing salvaged Tesla Battery Modules for solar energy storage, we reverse engineered the Tesla BMS communications protocol and developed a "controller" that would communicate with the Tesla BMS and control access to the batteries via contactors. It might be nice to add an optional display.

Andromeda offered other versions of the EVIC with interfaces or "skins" for the HPEVS AC motor and Ewert Systems ORION Battery Management System. The Orion version used the touch screen and the HPEVS did not and so Andromeda would send us the Orion version. We were replacing the software anyway for the Tesla Drive Units.

So we simply plugged in the EVIC and sent it CAN messages to drive the Orion display.. As the systems were different, we found many data pieces in the messages simply didn't apply to our system.

Andromeda Interfaces is dropping the EVIC product line. Ergo the development of the Raspberry Pi based EVTIV Battery Display. The CAN messages are simply legacy remnants. But they serve well enough.

You can do similarly. By sending these messages to the EVTIV Battery Display, either by CAN or by UDP Wireless packet, you can easily display most crucial battery data. And the EVTIV Battery Display performs logging functions that are quite useful as well.

MESSAGE ID 0x150

Length-8 bytes

Byte 0/1

Pack Current. LSB/MSB 16-bit signed integer containing pack current. Positive values are read as charging and negative values are read as battery discharge currents.

Byte 2/3

Pack Voltage. LSB/MSB 16-bit unsigned integer representing total pack voltage x 10. Multiply your reading by 10 to preserve one digit of decimal accuracy. Example: 48.76 volts sent as 487.

Byte 4/5

Pack Ampere-Hours. LSB/MSB signed integer representing ampere-hours used from pack x 10. Almost always a negative value. X 10 to preserve one digit of decimal fractional value. Example: 114.56 amperes discharged from pack – send as -1145.

Byte 6

Maximum Battery Temperature in whole degrees Celsius. 8-bit unsigned byte representing highest cell terminal temperature currently.

Byte 7

Minimum Battery Temperature in whole degrees Celsius. 8-bit unsigned integer representing lowest cell terminal temperature currently.

MESSAGE ID 0x650

Length-1 byte

Byte 0

SOC. 8-bit integer representing SOC*2. Take SOC, double it, and round it to a whole value. Example: $77.75\% \times 2 = 155.5$. Round to 155. On display we will divide by 2 to get 77.5. In this way, $\frac{1}{2}$ digit accuracy in 8-bits.

MESSAGE ID 0x651

Length-6 bytes

Byte 0/1

Low Cell Voltage. LSB/MSB 16-bit unsigned integer representing the lowest individual cell voltage in the pack *1000. Example: lowest cell = $3.214\text{v} \times 1000 = 3214$.

Byte 2/3

High Cell Voltage. LSB/MSB 16-bit unsigned integer representing the highest individual cell voltage in the pack *1000. Example: highest cell = $3.274\text{v} \times 1000 = 3274$.

Byte 4/5

Average Cell Voltage. LSB/MSB 16-bit unsigned integer representing the average if all cell voltages in the pack *1000. Example: average cell = $3.262\text{v} \times 1000 = 3262$.

MESSAGE ID 0x652

Length-8 bytes

Byte 4/5

High Voltage Cutoff. LSB/MSB 16-bit unsigned integer representing the highest individual cell voltage allowed before the battery system shuts down to prevent overcharge *10. Example: High Voltage Cutoff cell = $4.20\text{v} \times 10 = 42$. Displayed as the high voltage tick on the voltage dial.

Byte 6/7

Low Voltage Cutoff. LSB/MSB 16-bit unsigned integer representing the lowest individual cell voltage allowed before the battery system shuts down

to prevent overdischarge *10. Example: Low Voltage Cutoff cell = 3.25v x 10 = 32. Displayed as low voltage tick on voltage dial.

MESSAGE ID 0x68F

Length-8 bytes

Byte 0

Sequence Number. 8-bit unsigned integer. Message 0x68F provides individual cell voltage data for the entire battery pack. As a pack can vary from six cells to hundreds, it does this by sending a sequence of ordered 0x68F messages with six individual cell voltages in each message. As some messages may be dropped, this message provides both a sequence number and the number of six cell messages in total. In this way, we use the sequence number to retrieve the individual cell voltages in the correct order. This number will increment with each message until all cell voltages are sent, and then begin again at 0.

Byte 1

Total Modules. 8-bit unsigned integer indicating the total number of 6-cell messages in sequence. Derives from Tesla Model S modules which each have six cells.

Byte 2-7

Cell voltage. 8-bit integer. Each byte contains the voltage of one cell-2.00v. The result should be multiplied by 100 to retain two digits of decimal fractional accuracy. Example: $3.475\text{v} = 3.475 - 2.00 = 1.47 * 100 = 147$. The display adds 200 to that value = 347 and divides by 100 for 3.47v.

Note that the EVTV Battery Display actually graphs the voltage levels of all cells in the battery pack. As the number of cells varies from pack to pack, how does this work?

We use the Byte 1 data from the 0x68F message to determine the number of graph bars necessary, and scale the bar width appropriately. There are display limits to this and so the maximum MODULES displayed is limited to 40 or 240 individual cells.

MQTT AND THE AMAZON CLOUD

As described, the EVTV Battery Display can communicate with the EVTV ESP32 Battery Module Controller using either hardwired CAN or wireless UDP transmission through a local Internet Access Point or AP.

In practice, the CAN interface simply requires too much additional hardware and wiring to overcome the advantages of a wireless interface that allows your display to be anywhere within range of your AP. We use UDP wireless almost exclusively.

Current technology increasingly supports a global Internet of Things (IoT) approach to connecting devices.

History of the World – Part Duh. In 1999 IBM was working on a system for oil companies to monitor remote pipelines. To do so, they developed their MQ series or Message Queuing Series of devices and with that the **M**essage **Q**ueuing **T**elemetry **T**ransport protocol. This data protocol relied on the **T**ransport **C**ontrol **P**rotocol (**TCP**) which of course in turn relied on the **I**nternet **P**rotocol (**IP**).

It proved useful and a dozen years later was released as an open protocol anyone can use. **Message Queuing** always was a bit of a misnomer. It never precisely **QUEUED** anything in the normally accepted technical sense.

Amazon used MQTT for their popular ALEXA product. And Amazon has an endearing business model for productizing their internal tools in an increasingly useful Amazon Web Services (AWS) Cloud services suite so others can use them as well – for a fee of course. EVTV has hosted on AWS going back to 2009. It is a very clunky and difficult interface to do anything, but what it lacks in user friendliness it more than makes up for in global **SCALE** and durability - necessary to operate the world's largest online retail store.

That service today is Amazon's IoT Core service offering.

<https://aws.amazon.com/iot/?ft=n> Alexa has meanwhile evolved and an

additional AWS LAMBDA layer added to add services to Alexa. But you will see the similarities to MQTT immediately.

MQTT does a curious thing. It decouples the entire concept of the Internet and Internet “connections” with a man in the middle attack of gargantuan proportions.

IoT Core provides a forwarding agent termed a **BROKER**. And end points that use the service are **CLIENTS**. And this modifies the entire concept of client/server data handling. The Client doesn’t connect to a server, and a server doesn’t handle connections from clients. Instead, they both connect to an intermediary termed a **BROKER**.

The data itself becomes an object of very flexible size and format termed a **TOPIC**. Clients can either **PUBLISH** topics or **SUBSCRIBE** to topics, or both.

So the **BROKER** acts as a kind of anonymous post office/server where anyone can subscribe to and receive data of interest, and anyone can publish data of interest, and they don’t have to know anything about each other. The only connection is the **TOPIC** data objects. And with that, 10 million clients can all subscribe to the same topic as easily as one.

More of a cranial assault is the idea that one client could, for example, publish **TEN** topics on temperature, humidity, GPS position, battery voltage, etc. And any client could subscribe to **ONE** of those, all of them, or any of them. And indeed each client can do that differently depending on their needs.

A key element is that there **IS** a connection – that of any client to the broker. And AWS uses heavy X.509 256-bit public/private key encryption for the connections.

Amazon offers an amazing variety of other services in AWS and of course any of them can store this topic data in a database, present it on a web page, link to an Alexa “skill” etc.

And so this opens the possibility of computers and display devices such as iPads and iPhones and Androids to access data from “sensors” in refrigerators, door locks, air conditioners, wrist health monitors, or solar arrays and present it or otherwise use it.

And so you see how Alexa can turn on lights, lock your door, or manipulate your thermostat. Fitbit can collect information on how you are sleeping and present it on your phone. And a brazillion other applications.

As this will be an increasingly important method of tying the world together via Internet, we added MQTT capability to our ESP32 Tesla Battery Module controller and Battery Display more as a learning exercise than anything else.

Using MQTT is entirely optional. If you do use it, understand that EVTV could peer into your solar energy storage data as we provide the necessary Amazon IoT core broker for this to work. And no, the security credential certificate issues are too impossibly icky for us to make it where you can set up your own broker. In theory....well perhaps in the future....

Let's go back to our configuration screen. Note the **Use MQTT** check box. If you enable this, the system uses MQTT **INSTEAD** of UDP. If you uncheck it, it uses UDP through the local AP.

There are also two elements for IDs. The **BMS ID** is the important one. This is the ClientID of the ESP32 Battery Module Controller and is entered on the configuration screen of that device. The **BMS ID** here must match that exactly.

The **Display ID** can be anything. We just need a unique client ID for the display itself. Ensure you have a different client ID for each EVTV Battery Display you use.

SAVE those configuration items by touching the SAVE icon. Then power cycle the Display and on boot up, it will establish a connection to the EVTV broker on Amazon as the **Display ID**. It will also SUBSCRIBE to the topics

published by the BMS Controller as its client ID, in this case **601MorganOak**. That's EVTV's street address. But it could be anything you set in the BMS Controller configuration screen.

In this way, the **Bedroom1** display connects to Amazon's AWS IoT core EVTV broker and SUBSCRIBES to **EVTV/601MorganOak/Status** along with a series of topics providing individual cell data (**EVTV/601MorganOak/Module1**, **EVTV/601MorganOak/Module2**, ...**EVTV/601MorganOak/ModuleN**).

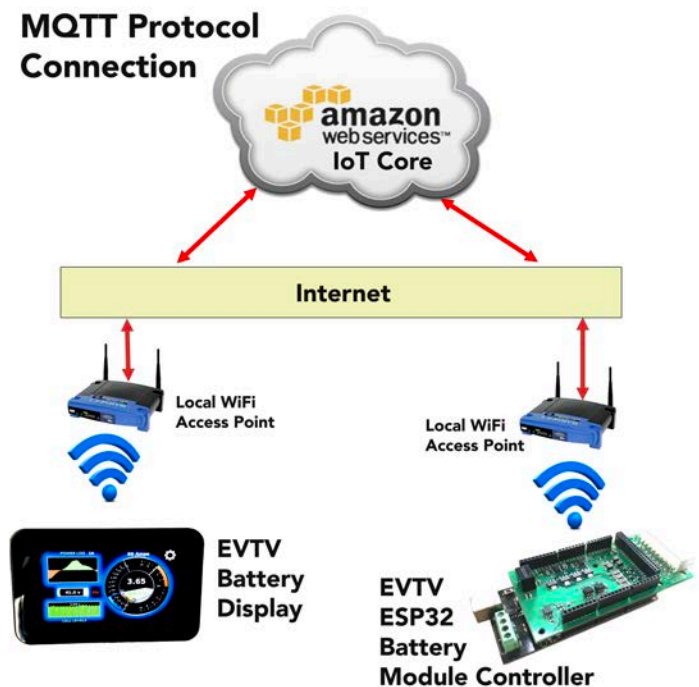
The ESP32 Battery Module Controller ALSO establishes a connection to Amazon IoT EVTV broker and PUBLISHES these topics.

Note that there IS no connection from the display to the battery controller. They each connect to the BROKER. And IF they both happen to be up and running, the data is passed and used.

We should note that the display is NOT as smooth and crisp as it is in UDP operation. It's a bit jerky and erratic. But surprisingly, it takes about 3 minutes to get used to that, and the nature of solar energy storage is that you don't really need smoothly instantaneous data. You really don't notice it after a few minutes.

And so of course, you see that that data can be displayed on the EVTV Battery Display. But actually it could be displayed on TWO of these displays. Or 25 million of them.

The immediate utility is you can have a display in your equipment room, another in the kitchen, and a third in the Republic of Sri Lanka and have a deep insight into the operation of your solar energy storage system, and so to your solar system and home power, from anywhere with an



Internet connected AP.

But it opens the door to a lot of future development. We can probably host a web site on AWS where you can do away with the display entirely, and view most of the important data using a web browser.

Obviously this lends itself to easy development of an iPhone or Android or iPad application. And off to the races in the future... You may one day simply ask Alexa what your current state of charge and power level is. Or what it is in your home in Costa Rica... The possibilities quickly overwhelm.

VERSION 1.2 UPDATE/CHANGES TO MQTT

With Version 1.2 of the EVTV Battery Display, we've introduced some interesting updates based on the obvious question everyone is going to have: "How do I use my OWN Amazon AWS IoT account?"

A corollary to this is "I'm very uneasy publishing my electricity usage data to EVTV."

We've wrestled with this extensively but believe we've come up with a credible solution. We no longer carry the traffic at all and to use MQTT you must set up your own Amazon AWS IoT Core account. This will make all your communications much more secure with us out of the loop.

1. We no longer publish via MQTT from the ESP32 BMS Controller. It communicates with EVTV Battery Display's only through the local wireless hub using UDP and TCP/IP.
2. Any display can receive and send through the local wireless hub to an ESP32 BMS Controller.
3. By electing **MQTT SENDER**, a display will echo the data received from the ESP32 BMS Controller to your Amazon IoT Core account REST API endpoint.
4. By electing **MQTT RECEIVER**, a second, third, or nth display can receive MQTT data from the **MQTT SENDER** via the Amazon Iot Core.
5. You must have at least one **MQTT SENDER** on the local wireless hub with the ESP32 BMS Controller in order to read any data on a remote **MQTT RECEIVER** display.
6. All displays can send configuration data which will be relayed to the ESP32 BMS Controller.

You can set up your own Amazon AWS Iot Core account using instructions at

<https://docs.aws.amazon.com/iot/latest/developerguide/iot-sdk-setup.html>.

In doing so, you will receive a connection URL and two TLS1.2 encryption certificates.

To use MQTT on a display, you must load the encryption certificates and a URL file onto the display.

You do this by placing the files on a FAT16 formatted USB thumb drive titled **BATTERY** under the following file names.

aws_url
aws_cert.pem.crt
aws_privkey.pem.key

The thumb drive **MUST** be titled **BATTERY**.

Then power cycle your display and the new files will be copied onto the display.

Amazon uses public–key cryptography to encrypt and decrypt login information. Public–key cryptography uses a public key to encrypt a piece of data, such as a password, then the recipient uses the private key to decrypt the data. The public and private keys are known as a *key pair*.

To log in to your instance, you must create a key pair, specify the name of the key pair when you launch the instance, and provide the private key when you connect to the instance. This enables you to securely access your instance using the private key instead of a password.

aws_url

This file simply contains the Uniform Resource Locator to connect to your Amazon AWS account Thing endpoint. Something like this:

```
a2718fm89xva84-ats.iot.us-east-1.amazonaws.com
```

This is the REST API endpoint needed to connect to your specific device shadow you created on Amazon's IoT Core. They provide this to you when you create a THING on your IoT account. It will of course be slightly different than this example and indeed is unique to each THING on the service.

aws_cert.pem.crt

This is the client certificate the Battery Display sends to AWS during the handshake once it has validated the server key. It is encoded with the AWS public key. AWS then uses it's private key to validate this certificate before beginning data transfer. In this way, Amazon can validate that it is communicating with a client who is appropriate for access to this specific IoT Core THING.

```
-----BEGIN CERTIFICATE-----
```

```
MIIDWTCCAkGgAwIBAgIUVOLk+r1tEbx1scwcXdzN3M1ar+owDQYJKoZIhvcNAQEL
BQAwTTFLEkGAlUECwxCQW1hem9uIFdlYiBTZXJ2aWNlcyBPQUGtYXpvi5jb20g
SW5jLiBMPVNL1YXR0bGUU1Q9V2FzaGlzR3RvbiBDPVVTMB4XDTE4MDkxODAwMDQw
M1oXDTQ5MTIzMTIzNTk1OVowHjEcmBoGB1UEAwTQVdTIElVVCBDZXJ0aWZpY2F0
ZTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBBAKcw5TjKhHYFmvc9b6Z2
aqTpogS73MfB4Wn7/kZGjKnIDylgD6YXFAtvtlQx68uwe5wONS9wXGjf2qEnijhc
cxky18L/94Nawy1MKSQWYaULAg+fCn14nOMMR1RAFm5NL253A/AIn0Vsk708EAMz
q3c+xu0stvvk870i+tP6VuKjuhQeTx4jPXIV06fiGVNy/9ySXcv8WHzkIjcJ3H/p
BEU0VY6SpbAPS4a2k12BiorBVEJcfMTlka9/ilt5Qg+VOuCM16eVegsEzR7vqyxh
+iZPSnXq0DE7zPw5NbA6gNE/rm/K22FTrrZQCs3ySLPgCAH5sdk+KESwPPJDUi6J
Z+OCAwEABaNgMF4wHwYDVR0jBBgwFoAUu2iZqsARB46JF5yuECg2uXGmXBkwhQYD
VR0OBBYEFHBF18+gV3VchLrnFX8jXSX+0MaIMAwGA1UdEwEB/wQCMABwDgYDVR0P
AQH/BOQDEgeAMA0GCSqGSIb3DQEBCwUAA4IBAQA/svUnY/IpcifTI9+ZKLQ6oNpL
CHgeYCBYzPTpKWJjRGhkMjh/2vzfx+m/8mluqYlP7YzCQp33ixKcyYEAZIbti7Xa
FIFUL83cc7WjrtBg5rh6TDmzCw8lUpyvxyHCl1qYBD7yotOivgUibA1SqOWfhCf/
cde9kAqVwMYTeNrfCfPz/hUctzU/z++QjIpN87c6iswqRMtc2SuCoNsxx6ajW1ld
17Q0IgoYkukY7Oy6egVgqqsNN8wxarVgM4dLB+1MLrgB+nTFQgg23eZzrMiUgRxH
mgXJ9k4vQOrlZ0lx/4oG1lvDmlzY29bKm5lvZfZgSr+GhIAYhkv6Aii6C2/r
```

```
-----END CERTIFICATE-----
```

aws_privkey.pem.key

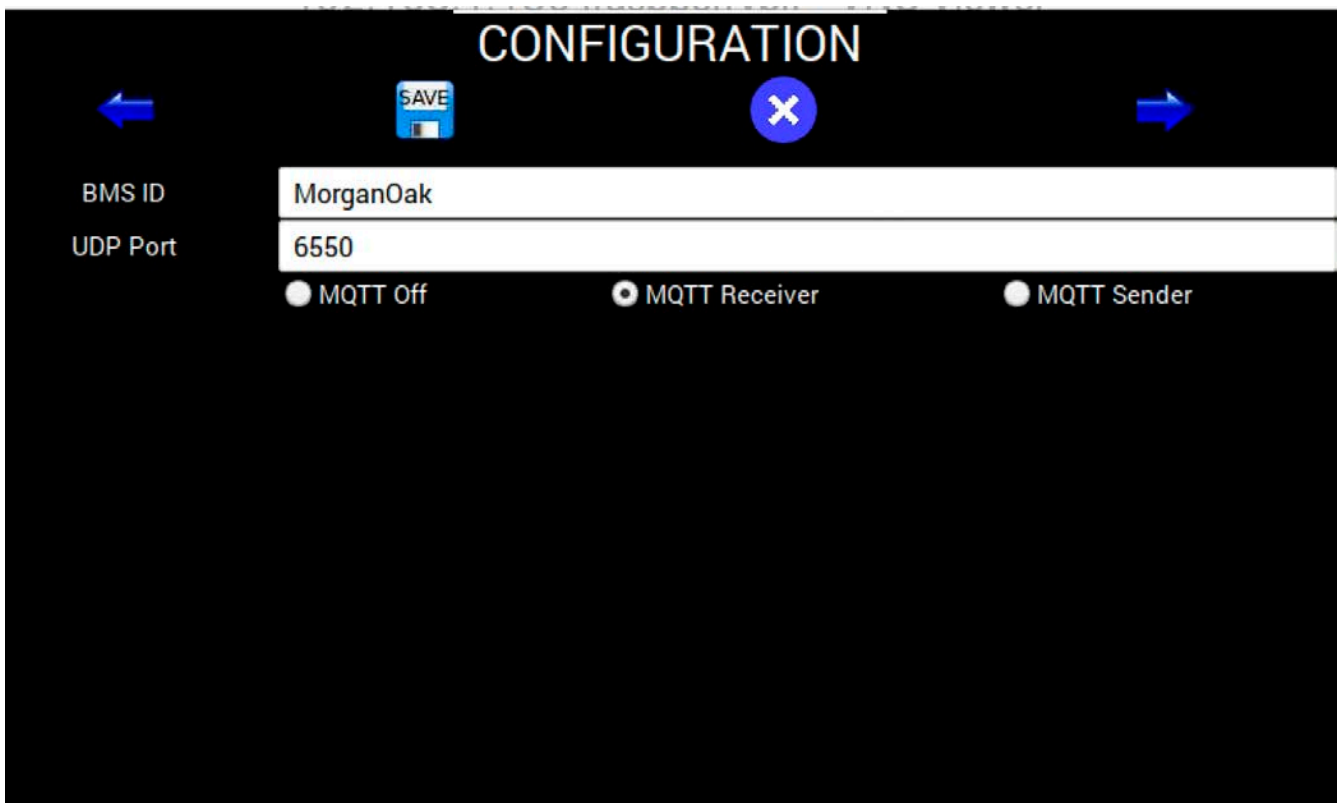
This is your private encryption key. Amazon will send messages encrypted with the public key. You need this private key to decode any messages from Amazon.

```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBACGCAQEApzDlOMqEdgWa9z1vpnZqpOmiBLvcx8Hhafv+RkaMqcgPKWAP
phcUC2+2VDHry7B7nA41L3BcaN/aoSeKOFxzGTLXwv/3g1rDLUwpJBZhpQsCD58K
fXic4stHVEAWbk0vbncD8AifRWwrs7wQAzOrdz7G7Sy2++Tzs6L60/pW4qO6FB5P
HiM9chXtp+IZU3L/3JJdy/xYfENiNwncf+kERTRVjpKlsA9LhraSYZGkisFUQ1x8
xOWRr3+KW3lCD5U64IzXp5V6CwTNHu+rLGH6Jk9KderQMTvM/Dk1sDqA0T+ub8rb
YVOutlAKzfJIs+AIafmx2T4oRLA88kNSLoln7QIDAQABAoIBADbdIrtKKcGZAUtU
y8iyXziSamoXQ9IBW3kuCjkBebNVRTRso3X7aMZ/+DMClq3W0hlZyDYzBwAbRWFQ
2li4bcS9HHSPPdqf6JsY8kduXxJ8mR5zcsdKOU9zonBWXmkTD62ayg4ZHLgzX/FdU
xWRnlkLhROIbpehWz1AJkYnyQoTIccQgfJcKnDiBn9+tPCTSwPo2VYIWGwVgPKN8
iek1PoNt7SR6ce09B/PKaq3X4vA5cazUSLTIqym75qIX/cPFycYnFgY3UInq7zhZ
fKsrKuvOIGzveqDYEBwIk9HxOyEw1UVTJ1/iDw3aagsP+wZXOYReCLtDANsmhsD
2aCExUECgYEAOPyO1L4s58MQQLozQT83gVhZU8d5/ZZ+D2U5K9vsQvQmqyJbazIp
yEejXuylnuyOU8fCk3WQVDAG5uLRPsg7iaeuUgFdW8QAP51XxpOuVh/1oEv7deVO
4sTYfaBxhcNqhxEVVri4Gw59vpnS43JnNo13sWEVM5XSBcrgl+xlukCgYEAzM4Z
nh64kp8yO1+cj+sbQvBpUtzsGA4kV2NPwJhmFS6HDWigMKnDErknjPQHleeFbI17
NJmGJB1TKVcCVuSOK0JV1I+cMfmmef4lDuTvRucgW5ZIfhM7ZXEaNBdwQwrVVriH
bN5vEdvxJzaZKzvHbn4z5p1l7Mb3oG8osT6jLmUCgYB/e7D9Yc0uy7UcZvlhMdkA
FA7ZpnNqF+VqKstHT+69oQX9mwW9TGpkfB7ShvU5DwB7Zv3wyeRzFqD/MxmFfMuk
2x7hc2ep1NR5+ddTkt711rUVRyRtlKYcewu9BqR5fo17OxaSRHP6PUrDfD5C9MT1
aHK9R4WRZDEry2WcyQciEQKBgAWkywluc0unE21LbzcQFY08scpUyDTJfvrhg7sc
+AMxawtVjMJKiROlApAB0Y1lednJ2BlU+btZC4eiHDBkz/41L9K0OpGt/YZzGHZW
nLlpeRzNAWZoBUUAXuZ8ltliGQrwtqjLpBhbT3VZVcpqI9E7w01wPdLbFuBdK+d
uEV5AoGBAI+WLZhEW7hi6it4v4xpZ0nmjCQs1orRA9k2Pa01rnVtfr08Inzglw/J
FzZk+MHvYiS6e8NS18JosPYiR1NoyhBA1cvMzhgdK4CjwGJsLkMxprqV4tRJNPiB
WcehS+xSzluDdQAsXYaEEVBuTVa51+NwodAm59YN/6+3byeNe7Uj
-----END RSA PRIVATE KEY-----
```

Amazon also creates these files and allows you to download them when you set up your account. You need only copy them into your display.

Amazon provides free sandbox type of accounts with a small data allowance of 250,000 messages per month. You can later upgrade this to a paid account if your usage exceeds the basic allowance.

In this way, you can achieve global MQTT communications using multiple display devices, have your MQTT data secured by encryption, and have it on an account YOU own and control.



This is the new configuration screen for the MQTT functions.

MQTT OFF

With **MQTT OFF** the display will receive data only over UDP from an ESP32 BMS Controller. Here, the **UDP PORT** variable becomes important. You can have multiple ESP32 BMS Controllers and Battery Monitor Displays on the same wireless hub by pairing them to use the same **UDP PORT**. The default is port **6500**.

MQTT SENDER

With **MQTT SENDER** selected, you will **STILL** receive data from the ESP32 BMS Controller paired using the **UDP PORT**. But this data will also be relayed and published as a topic on the Amazon IoT Core service.

As a result, a sender **MUST** be on the same wireless hub as the ESP32 BMS Controller in order to receive the original data. And that hub must be connected to the Internet to publish/relay the data to the Amazon service.

You must have a local sender for any remote displays to work as a receiver.

The topic published will be individually identified using the contents of the **BMS ID** field.

In this way, you can have more than one ESP32 BMS Controller, and more than one **MQTT SENDER** on the same local wireless hub. The data is differentiated on Amazon IoT Core by **BMS ID**.

MQTT RECEIVER

If **MQTT RECEIVER** is selected, no UDP data will be received and the **UDP PORT** setting doesn't matter.

BMS ID identifies the topic subscribed to. As a result, this variable pairs your remote display to the specific **MQTT SENDER** using that **BMS ID**.

HOW MUCH DOES AMAZON AWS IOT CORE COST?

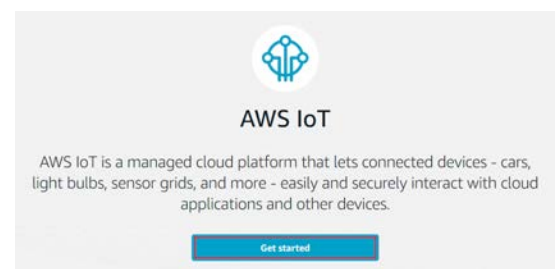
Amazon offers free accounts for 12 months on IoT Core with a limit of 250,000 messages per month. Unfortunately, that's about one day's worth of Battery Display traffic.

With one Battery Display configured as a MQTT SENDER and a second display as a MQTT RECEIVER, you will generate about 10,368,000 messages per month. Currently Amazon pricing is \$1 per million messages so about \$11 per month.

The Battery Display sends an MQTT update message every 500 ms or twice per second.

TO CREATE AN AMAZON AWS IOT CORE ACCOUNT

1. Log into Amazon AWS. You may need to create an account with AWS.
2. Select **SERVICES** at the top of screen.
3. Select **AWS IoT Core**.
4. On the left menu, **Get Started** (Only necessary if you've never used IoT before)
5. At the bottom left select **Settings**
6. A **Custom end point URL** will be displayed. Copy this to your clipboard.
7. Create a file on your PC named **aws_url** and paste the custom endpoint URL as the first (and only) line of this text file. Copy this file to a USB thumb drive named **BATTERY**.



8. Select **Secure** on left hand menu
9. Select **Policies**
10. Select **Create Policy**.
11. Enter a NAME for the policy in the field provided.
12. Set **Action: "iot:*"**
13. Set **Resource ARN: "*"**
14. In **effect** select **Allow**
15. Select **Create** button in lower right
16. Select **Certificates** under **Secure**
17. Select **Create a certificate**
18. Under **One-click certificate creation** select **Create Certificate**
19. Select **Activate**

The server will create three files that look something like this, but file names vary with each THING.

ce8f02be46.cert.pem - The certificate for authority.

ce8f02be46.public.key - Your public key. You don't need this, Amazon uses it

ce8f02be46.private.key - Your private key. You use this to validate

20. Click the **download** button next to **ce8f02be46.cert.pem**

21. On your PC, rename this file **aws_cert.pem.crt** and save it to the thumb drive named **BATTERY**

22. Click the **download** button next to **ce8f02be46.private.key**
23. On your PC, rename this file **aws_privkey.pem.key** and save it to the thumb drive named **BATTERY**
24. Select **Attach a Policy** and check the box for the policy you created.
25. Select **Done** to complete the setup of your AWS IoT THING.

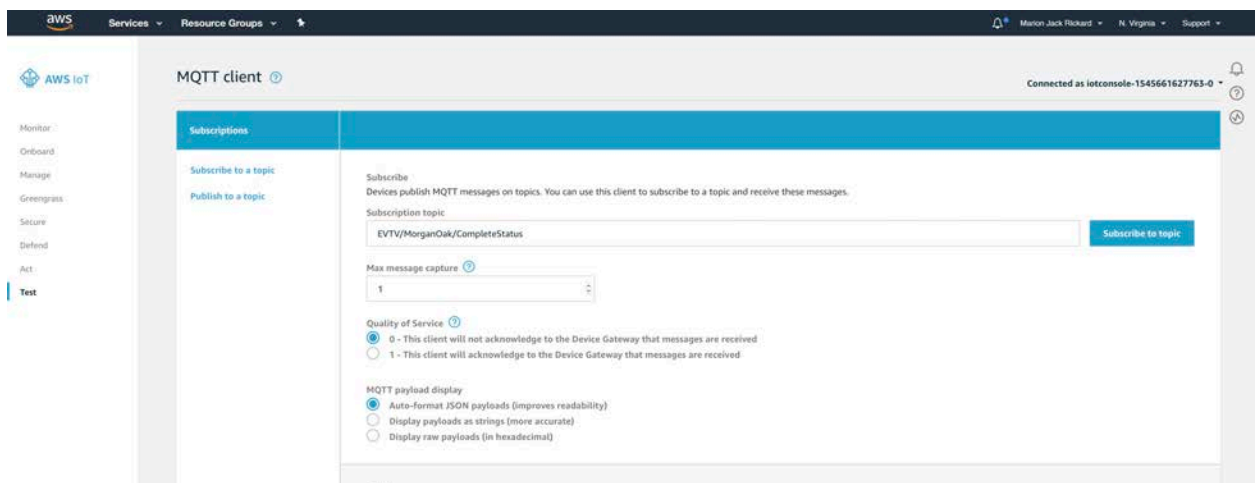
At this point, you should have an operating AWS IoT Core THING set up and a USB thumb drive named **BATTERY** with the three files on it.

Insert the thumb drive into any Battery Display USB port and power up the display. Once the Battery Display is up and running **YOU MAY REMOVE THE BATTERY THUMB DRIVE**. The files are copied into a local directory if they are found on powerup.

On the configuration screen, make sure the UDP Port matches the port set on the ESP32 BMS Controller and that the device is displaying data from the battery.

Select **MQTT SENDER** and **SAVE** settings. The Battery Display should be receiving and displaying battery data over UDP and now sending MQTT packets to your Amazon end point specified in **AWS_URL**.

You can confirm this on the Amazon service.

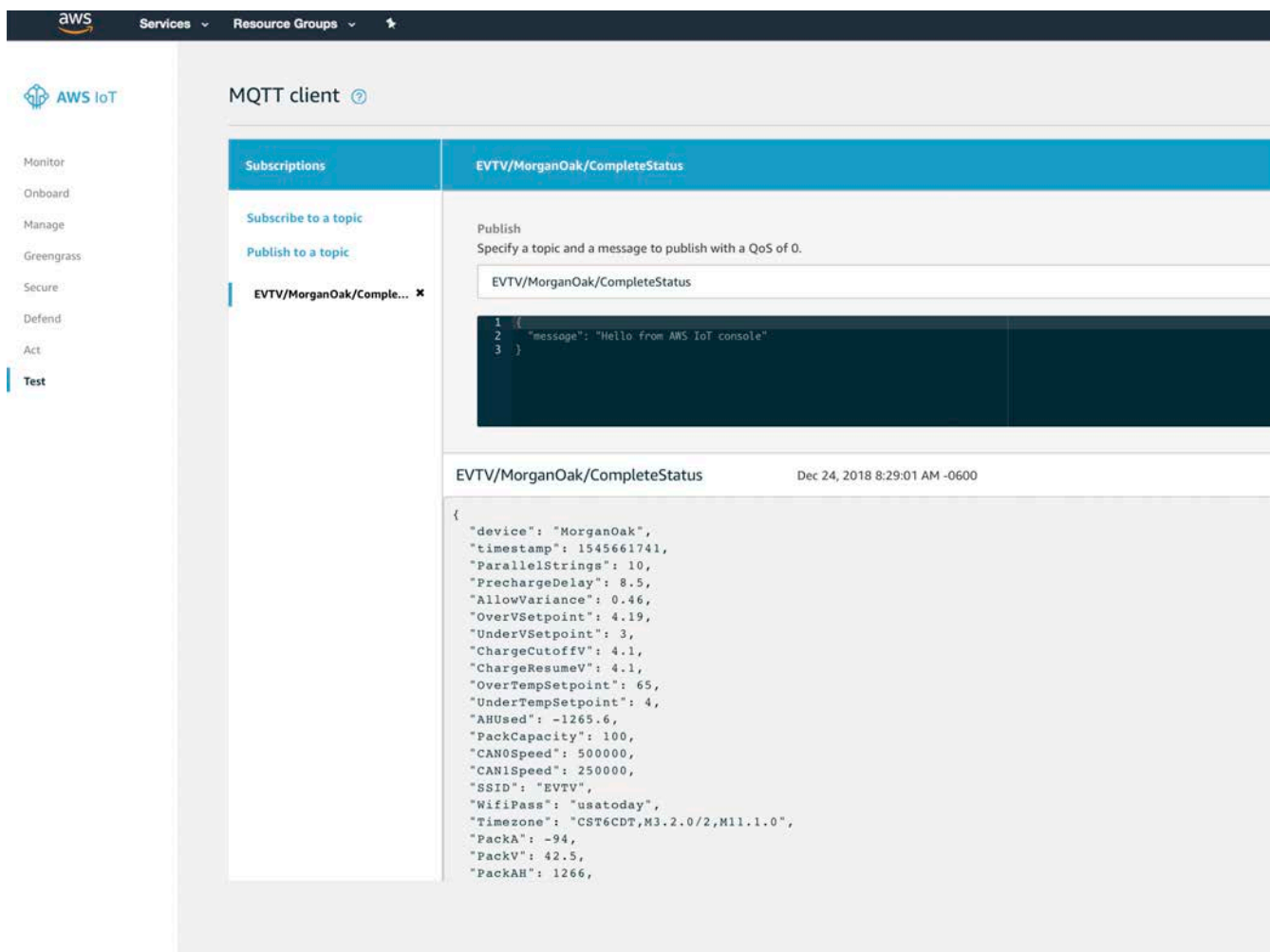


Login to your account and select AWS IoT. Then select the last item on the menu on the left of the screen – **TEST**.

There is a field provided for you to enter a topic to subscribe. The topic published by the Battery Display Unit will be **EVTV/YourBMSID/CompleteStatus**. Enter that in its entirety. You can also enter the number of messages to receive and for this enter **1**. Then select **SUBSCRIBE TO TOPIC**.

The center section of the topic subscribed is shown here as **YourBMSID**. This should of course correspond to the **BMS ID** entered on the Battery Display configuration screen.

Once you have “subscribed” to the topic it will be displayed and updated as new messages are received.



The screenshot displays the AWS IoT console's MQTT client interface. On the left, a navigation menu includes 'Monitor', 'Onboard', 'Manage', 'Greengrass', 'Secure', 'Defend', 'Act', and 'Test'. The main area is titled 'MQTT client' and features a 'Subscriptions' panel on the left with options to 'Subscribe to a topic' and 'Publish to a topic'. A subscription is active for the topic 'EVTV/MorganOak/Comple...'. The right panel shows the 'Publish' section with a text input field containing 'EVTV/MorganOak/CompleteStatus' and a code editor with a JSON message:

```
1 {
2   "message": "Hello from AWS IoT console"
3 }
```

. Below this, a received message is shown for the same topic, timestamped 'Dec 24, 2018 8:29:01 AM -0600'. The message is a JSON object with various fields:

```
{
  "device": "MorganOak",
  "timestamp": 1545661741,
  "ParallelStrings": 10,
  "PrechargeDelay": 8.5,
  "AllowVariance": 0.46,
  "OverVSetpoint": 4.19,
  "UnderVSetpoint": 3,
  "ChargeCutoffV": 4.1,
  "ChargeResumeV": 4.1,
  "OverTempSetpoint": 65,
  "UnderTempSetpoint": 4,
  "AHUsed": -1265.6,
  "PackCapacity": 100,
  "CAN0Speed": 500000,
  "CAN1Speed": 250000,
  "SSID": "EVTV",
  "WifiPass": "usatoday",
  "Timezone": "CST6CDT,M3.2.0/2,M11.1.0",
  "PackA": -94,
  "PackV": 42.5,
  "PackAH": 1266,
}
```


You will note the “**timestamp**” in the data display updates regularly.

This topic contains quite a bit of data about your battery system. We basically put everything but the kitchen sink in one topic for a reason.

Amazon bills based on the rate of \$1 per million messages. But messages can be up to 5kb in length. So it is less expensive to put all data in a single message rather than many brief messages.

Once you have your basic Battery Display acting to receive data on a matching UDP port from the ESP32 BMS Controller, and publishing MQTT messages as an MQTT SENDER with your BMS ID, you can then set ANY number of additional Battery Displays up as MQTT RECEIVER. But note that that display will need to load the same three files from the BATTERY thumb drive at least once on powerup, and have the same BMS ID entered on its configuration screen. It will then display the same BMS data echoed through the AWS Iot Core.

It is important to note that the **MQTT SENDER** battery display must be on the same wireless hub as the ESP32 BMS Controller so it can receive data over UDP.

But you can have any number of **MQTT RECEIVER** displays, and each connected to ANY wireless hub with Internet connectivity anywhere in the world.

MORE ON WIRELESS HUB CONNECTIONS

Generally, once at the Raspian desktop, you can do anything with the display that you can on any Raspberry Pi Model 3B+.

Once you have connected to your wireless hub once, the Display will remember that connection and make it automatically any time it powers on.

However, it may be that the DHCP server on your wireless hub assigns a new IP number each time. You might use VNC Viewer to view your battery display or perhaps you want to connect to the included web server on the Pi and desire a permanent “static” IP number for your device.

1. Click EXIT on the Battery Display configuration screen to drop to the Raspian desktop.
2. Touch the square black TERMINAL icon on the tools bar at the top of the screen to drop to the command line.
3. Enter **sudo nano /etc/dhcpd.conf** on the command line.
4. Make additions to the file as shown below:
5. Press **CTRL-O** to write out the modified file.
6. Press **CTRL-X** to exit the NANO editor.

```
interface wlan0  
static ip_address=192.168.1.72  
static routers=192.168.1.1  
static domain_name_servers=8.8.8.8
```

The first line identifies the wireless interface, in all cases WLAN0.

The second line contains the static IP number you want your display to have. In this case **192.168.1.72**.

The third line is the address of your wireless hub router, in this case **192.168.1.1**.

The fourth line should contain your domain name service server address. In this case Google's servers are listed as **8.8.8.8**

To make sure your current wireless hub is the default:

```
sudo nano /etc/wpa_supplicant/wpa_supplicant.conf
```

1. Modify the file as below:
2. Again, CTRL-O to write and CTRL-X to exist nano.

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
```

```
update_config=1
```

```
country=US
```

```
network={
```

```
    ssid="riverhouse"
```

```
    psk="usatoday"
```

```
    key_mgmt=WPA-PSK
```

```
}
```

Of course, modify the **SSID** and **PSK** to reflect your wireless hub and password.

AN INTERESTING WEBSITE TO VISIT

The Battery Display has another interesting feature. If you check the box on the configuration screen **LOG TO DATABASE**, displayed values are also logged to a mySQL database on the Raspberry Pi.

Additionally, a historical graphing program titled GRAFANA has been installed. Grafana uses data from the mySQL database to create user definable historical graphs. We have designed a series of these graphs just for the battery display. And the Raspberry Pi has the ability to serve as an Internet World Wide Web server at the same time it is displaying your normal battery data.

To connect to it: <http://192.168.1.72:3000> from your browser.

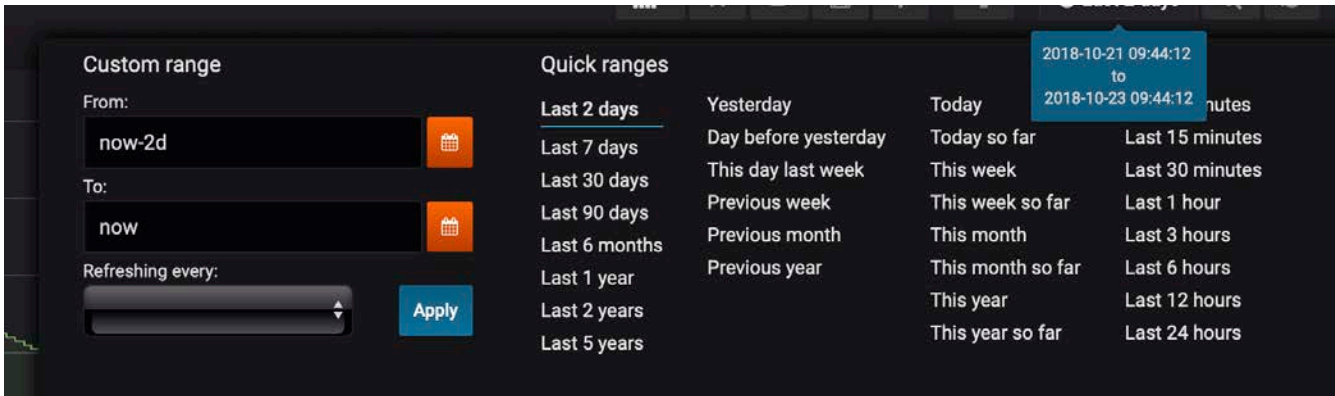
This assumes the IP number of your Display is 192.168.1.72 of course. By connecting to port 3000, you can access the internal Grafana web server.

<http://192.168.1.72:3000/d/vRg76b1mz/solar-stats-dashboard?orgId=1&from=now-2d&to=now>

This link shows all your graphs for the last two days.



If you click on the time specification in the upper right hand corner of the screen, you will get a menu of different time spans you can select to view data on this graph set.



On each graph, if you SHIFT-click on the bar at the top of the graph and then click VIEW, you will put that graph to a full screen view.



The Grafana interface is awkward but powerful. It is quite normal to have to wait 10-12 seconds for an update. But it has a great ability to allow you to examine the operation of your solar storage system over time in great detail. It is worth exploring this interface to become familiar with it.

SCREEN SHARING

Some configuration items on the Battery Display require you to enter short strings of text. A small wireless keyboard is provided to enable this. For example, you must select your home wireless hub in setting up your display and that usually requires you to enter a password. You can use this keyboard to enter that.

But once you are on your wireless system, there IS an easier way.

Virtual Network Computing (VNC) is a graphical desktop sharing system that uses the Remote Frame Buffering Protocol to remotely control another computer. It transmits keyboard and mouse events from one computer to another, relaying the graphical screen updates back in the other direction, over a network.

VNC is platform-independent – there are clients and servers for many GUI-based operating systems. Multiple clients may connect to a VNC server at the same time. Popular uses for this technology include remote technical support and accessing files on your work computer from your home computer, or vice versa.

VNC was originally developed at the Olivetti & Oracle Research Lab in Cambridge, United Kingdom. The original VNC source code and many modern derivatives are open source under the GNU General Public License.

The Battery Display has VNC Server already installed on it using the remote framebuffer protocol.

In 2002, AT&T closed their Cambridge, UK research laboratory, formerly the Olivetti Research Laboratory, ending a productive, decade-long collaboration with the world-famous Computing Laboratory at the University of Cambridge.

The original inventors of VNC®, immediately formed RealVNC® to build upon the enormous popular appeal of screen sharing technology with the open source community, and to leverage the growing commercial appetite for enterprise-class remote access software.

The VNC authors documented the protocol for this in an informational Internet RFC standard 6143. RFB ("remote framebuffer") is a simple protocol for remote access to graphical user interfaces that allows a client to view and control a window system on another computer. Because it works at the framebuffer level, RFB is applicable to all windowing systems and applications. This document describes the protocol used to communicate between an RFB client and RFB server. RFB is the protocol used in VNC.

So to access your Battery Display from your laptop or desktop computer, simply download and install the free VNC Viewer program for your particular operating system.

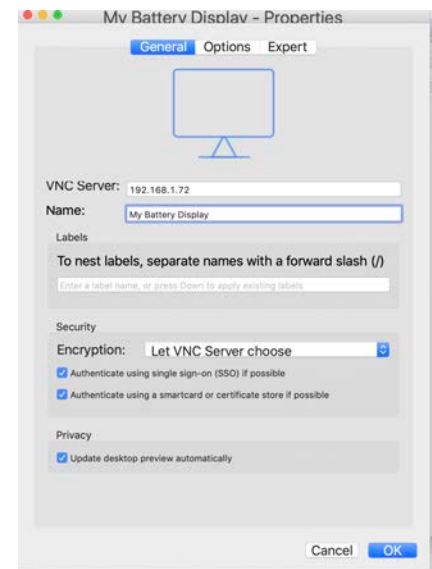
<https://www.realvnc.com/en/connect/download/viewer/>

When you run the program, select FILE and NEW CONNECTION.

You can then enter the IP number of your battery display (appears on the main screen).

Once you have “created” a new connection you simply click on it thereafter to “connect” to your battery display.

The FIRST time you do this it will prompt you to enter a USERNAME and PASSWORD.



Use “pi” for the username and “**evtv**” for the password.

