Grand Plan Outline

Author: William Kerr

# Introduction:

Goals:

My goal is to create a solar-powered greenhouse. The requirements of the greenhouse are to store plants at a reasonable temperature, store a monitoring system inside, and power said monitoring system with rooftop solar panels providing power.

UCSC has 24 solar panels that provide 1.5kWhr of power per day. The power will be delivered to the system properly using a MPPT Solar Charge Controller, provided by Morningstar. Each charge controller can only take 150V, and the total amount of solar panels we have sums up to 196V. Therefore, we will have two charge controllers working in tandem to charge a battery pack. These charge controllers can charge the same battery pack without interference, according to Morningstar. Both charge controllers will have a remote temperature sensor connected to it in order to monitor the amount of charge being delivered to the batteries. The charge controllers will deliver power to a battery pack, but these charge controllers cannot regulate all the batteries by themselves.

The battery pack consists of 8 batteries, each with a maximum charge of 180Ah and nominal voltage of 3.2V, provided by CALB. The batteries will be connected in series in groups of four, summing up to 12.8V per group. The battery pack will be stabilized, its output current regulated, and its temperature regulated by a Battery Management System (BMS), provided by EMUS. The BMS consists of a central control module, two CAN Cell Group Modules, two top isolators, two bottom isolators, and 8 Cell Isolator boards. All components will be controlled by a CAN bus originating from the central control module, that EMUS has programmed for its end users.

The BMS will control the charging of the batteries using variables such as the voltage, current, temperature, and charge of each individual cell, and modify internal variables such as the battery balancing rate accordingly to prevent damage to the batteries. If an individual battery dies, the BMS system will alert you with its monitoring system. The monitoring system can monitor from each battery: current drawn, individual voltage, temperature, and more. The monitoring system will calculate how much charge the battery has based on its own internal logic system. If the charge of any battery gets too low, it will focus the incoming charge from the solar panels on that particular battery. If the maximum charge of any battery gets too low, it will let you know with an alert. The monitoring system will monitor the batteries’ temperature. The BMS will cool the batteries if they get too hot, and warm the batteries if they get too cold. The monitoring system will let you know with an alert. There are other variables the monitoring system monitors. Check the components document for details.

The battery pack will power the internal computer system inside the greenhouse, an AC outlet somewhere in the greenhouse, and the 12V distribution nodes scattered throughout the greenhouse. Battery power will be converted from DC to AC using a power converter called the Suresine inverter, provided by Morningstar, that will be located near the Charge Controller. The AC power outlet will have a maximum power output of 300W. The 12V distribution nodes will distribute power to various devices that one might need for a greenhouse experiment, such as an air pump, hanging lights, a heater, a water pump, etcetera.

The internal monitoring system consists of a Raspberry Pi main microcontroller, which will be located next to the Charge Controllers for convenience, a 2G GSM shield attached to the main microcontroller, a 7” touch screen for convenience, the faculty sensors, the student sensors, and the Raspberry Pi slave microcontroller(s).

The faculty sensors will need to monitor light, humidity, and temperature, as per the client request. This can be achieved with light sensors and humidity & temperature combination sensors. Since I have the liberty to choose which sensors I will use, I will choose sensors that utilize the I2C serial protocol. This way, a daughterboard can be designed that will enable the main microcontroller to monitor up to 8 I2C devices. The daughterboard will have detachable inputs, so it is possible to change the sensors utilized in the greenhouse.

The main Raspberry Pi microcontroller is powerful enough to suit our needs. It will be running a Linux system called Raspbian. I will install a touch screen display to the main microcontroller, so if a faculty member needs to debug it, they can see what they’re doing without needing to connect to the Raspberry Pi using an SSH connection. In case of an emergency, I will include a way to connect to the main Raspberry Pi using an SSH connection and the proper permissions. The main Raspberry Pi and the touch screen will have an enclosure to protect it from the elements. The main Raspberry Pi’s tasks will consist of reading the sensor values from the faculty sensors using the I2C serial protocol, receiving a student sensor value package from the slave computers located around the greenhouse over the Bluetooth protocol, reading and storing the values generated from the Charge Controller’s internal PLC, reading and storing the values generated from the Battery Management System’s internal logic system, compiling all of that data, and then sending it over a 2G TCP connection directly to a server somewhere at UCSC with a GSM/GPRS daughterboard attachment. The 2G connection will be provided by Ting, as per client request.

The student sensors can consist of almost any type of sensor, and it seems random at this point in time. Here are some possibilities I can envision: water temperature sensors, soil moisture sensors, light sensors, humidity & temperature combination sensors, pressure sensors. Since it is impossible to know what serial protocol the student sensors will utilize, I will incorporate the most popular serial protocols into a daughterboard I will design, including I2C, SPI, UART, and 1-wire. I will incorporate a lot of UART ports into my daughterboard design, so that way if their sensor uses a strange protocol not listed here, a faculty member or another engineering student can make a bridge from that protocol to UART, write some code to read their sensor, and the slave microcontroller will be able to include their sensor in the data packet it sends to the master microcontroller.

The slave Raspberry Pi microcontrollers will be powerful enough to send data packets to other microcontrollers using the Bluetooth protocol, read sensor values over a variety of serial protocols, and process that data and compile it into a data packet. The slave Raspberry Pi will have an enclosure to protect it from the elements, but it will not have a sensor. It will be programmable from the main Raspberry Pi. The data packets will contain this type of data: name of owner, name of sensor, type of sensor, protocol utilized, value itself, what unit it’s measured in, time and date of when the value was taken.

The main Raspberry Pi microcontroller will be sending a packet of data, utilizing the GSM daughterboard, to a UCSC server somewhere. Once the server receives the data, it will store it in a SQL-style database. That server will also be connected to the internet, so users can connect to it from their phones. When a user connects to this server, the server will serve a website back to the user. The user can then use that website to send a request to the server asking for student sensor data, faculty sensor data, Battery Management data, or Charge Controller data. The user can add a format to their request, such as a graph over time, instantaneous data, data from the last time the computer has polled, data from a specific point in time, etc.

If the user requests a graph over time, they will need to specify what type of data they are requesting, Y-Axis units (i.e. V, mA, etc.), X-axis data (i.e. units of time), and range and scale for both X and Y axis for each dataset they request. Optional: a caption, a title, names for the X and Y axis. The server will return a webpage containing their graph, and some graph metadata: how many data points are returned, what the scale and range are, and a legend if they requested multiple data sets.

# Sources:

The Charge controller can work in tandem with another charge controller of the same brand:

<https://www.morningstarcorp.com/parallel-charging-using-multiple-controllers-separate-pv-arrays/>

Morningstar Monitoring Code Design Document

# Morningstar.py Design Document

Author: William Kerr

# Introduction:

The TSMPPT has a Programmable Logic Controller (PLC) inside. The PLC can be reprogrammed to respond to different battery voltages. Example: when the battery goes to 3.7V, the battery goes to the equalize stage. To reprogram the Tristar MPPT, you must use a PC and a RS-232 to USB cable, and the program MSView, provided by Morningstar. Link: <https://www.morningstarcorp.com/msview/>. I also have a document called Viewing Data on Morningstar Devices that explains how to use the software.

However, the Tristar MPPT can be monitored by any device capable of serial monitoring, such asa a Raspberry Pi. Morningstar.py will contain the code necessary to monitor data from a Tristar Morningstar MPPT solar charge controller.

Morningstar has a MODBUS specification document for the Tristar Morningstar MPPT. It should be in my App Notes section. If not, here’s a link:

Link: <https://www.stellavolta.com/content/MSCTSModbusCommunication.pdf>

Morningstar.py contains a class that reads PLC data, denoted as Morningstar(). This class can monitor data by reading it and dumping the data to a JSON file. Details are in the upcoming sections.

## Dependencies:

## modbus-tk:

The Tristar MPPT uses a royalty-free serial protocol called MODBUS. There exist many libraries to read it. The Python language has pymodbus, and modbus-tk. Since pymodbus is not reliable (and I want reliable code), I will be using the modbus-tk library. It is distributed under the GNU-LGPL license (GNU Lesser General Public License) © 2009. Created by Luc Jean – [luc.jean@gmail.com](mailto:luc.jean@gmail.com) and Apidev – [http://www.apidev.fr](http://www.apidev.fr/). No warranty of any kind.

### json:

Comes with every distribution of python. Necessary to convert dictionaries into JSON format and dump it directly to an outfile.

### serial:

This is a serial library for Python. It’s easy to use, and free. Provided as-is. Install with pip install pyserial. Use by calling “import serial”. ©2015 Chris Liechi [clichi@gmx.net](mailto:clichi@gmx.net) All Rights Reserved.

# Initialization

The Morningstar() class (like most other classes) has an \_\_init\_\_() function, that calls itself whenever a Morningstar() object is created. It requires port, baudrate, and the MODBUS slave number as arguments. When a Morningstar() object is created, it will initialize the serial connection to the PLC using this information. Then, it will create a MODBUS RtuMaster() class from the MODBUS-TK library.

After the MODBUS RtuMaster() class is initialized, it will call the internal function .scaling(). It will test what scaling factors are used.

Dictionaries in Python are everywhere in this code.

It will have methods to either dump data to the command line or dump data to an outfile. The outfile should be a .json file, since the contents will be written in JSON format.

# Classes:

## Morningstar()

### Description:

Reads data from the Tristar MPPT PLC. Takes PORT, BAUDRATE, and SLAVE\_NUMBER upon initialization.

### Variables:

PORT: what port number are you using?

BAUD: what baudrate are communicating at?

SLAVE\_NUMBER: what is the MODBUS slave number you’re reading from?

serial\_connection: the serial connection from pyserial that actually communicates with the PLC.

master: the class from modbus\_tk that converts serial data into data we can read.

### Methods:

**.scaling()**: Sets the classes internal scaling properties (V\_PU and I\_PU). Also prints the current running version to the console. Runs every time an object of this class is created.

**.ADCdata()**: Returns a dictionary containing ‘battery voltage’, ‘battery terminal voltage’, ‘battery sense voltage’, ‘array voltage’, ‘battery current’, ‘array current’, ‘12V supply’, ‘3V supply’, ‘meterbus voltrage’, ‘1.8V supply’, and ‘reference voltage’.

**.TemperatureData()**: Returns a dictionary containing ‘heatsink temperature’, ‘RTS temperature’, and ‘battery regulation temperature’. All are in degrees Celsius.

**.StatusData()**: Returns a dictionary containing ‘battery\_voltage’, ‘charging\_current’, ‘minimum battery voltage’, ‘maximum battery voltage’, ‘hour meter’, a list of faults, a list of alarms, the current state of the DIP switch, and the current state of the LED.

**.ChargerData()**: Returns a dictionary containing ‘Charge State’, ‘target regulation voltage’, ‘Ah Charge Resettable’, ‘Ah Charge Total’, ‘kWhr Charge Resettable’, and ‘kWhr Charge Total’.

**.MPPTData()**: Returns a dictionary containing: ‘output power’, ‘input power’, ‘max power of last sweep’, ‘Vmp of last sweep’, and ‘Voc of last sweep’.

**.Logger\_TodaysValues()**: Returns a dictionary containing: ‘Battery Voltage Minimum Daily’, ‘Battery Voltage Maximum Daily’, ‘Input Voltage Maximum Daily’, ‘Amp Hours Accumulated Daily’, ‘Watt hours accumulated daily’, ‘Maximum power output daily’, ‘Minimum temperature daily’, ‘Maximum Temperature Daily’, a list of daily faults, a list of daily alarms, ‘time\_ab\_daily’, ‘time\_eq\_daily’, and ‘time\_fl\_daily’.

**.ChargeSettings()**: Returns a dictionary containing: 'EV\_absorp', 'EV\_float', 'Et\_absorp', ‘Et\_absorp\_ext', 'EV\_absorp\_ext', 'EV\_float\_cancel', 'Et\_float\_exit\_cum', 'EV\_eq'], 'Et\_eqcalendar', 'Et\_eq\_above', 'Et\_eq\_reg', 'Et\_battery\_service', 'EV\_tempcomp', 'EV\_hvd', 'EV\_hvr', 'Evb\_ref\_lim', 'ETb\_max', 'ETb\_min', 'EV\_soc\_g\_gy', 'EV\_soc\_gy\_y', 'EV\_soc\_y\_yr', 'EV\_soc\_yr\_r', 'Elb\_lim', 'EVa\_ref\_fixed\_init', 'Eva\_ref\_fixed\_pet\_init'

**.DumpInstantenousDataToJSONFile(outfile)**: Calls all instantaneous data internal class methods (ADCdata(), TemperatureData(), StatusData(), ChargerData(), MPPTData()), and dumps them into an outfile using json.dumps(). Preferably, the file’s name will end in “.json” so the operating system can recognize that the file is in JSON format.

**.DumpDailyDataToJSONFile(outfile)**: Calls all daily data internal class methods (Logger\_TodaysValues() and ChargeSettings()) and dumps them into an outfile using json.dumps(). Preferably, the file’s name will end in “.json” so the operating system can recognize that the file is in JSON format.

BMS Monitoring Code Design Document

# BMS.py Design Document

Author: William Kerr

# Introduction:

The Battery Management System (BMS) has a Programmable Logic Controller (PLC) in it. It can be used to monitor things such as charge dissipated, voltage levels of each individual cell, etc.

The internal PLC can be monitored using a USB interface. This is what the Raspberry Pi will do using the interpreted Python Language.

The Python language is dependent on classes to process data. So, I will be writing a Python class to extract data from the BMS PLC.

## Dependencies:

### pyserial:

This is a serial library for Python. It’s easy to use, and free. Provided as-is. Install with pip install pyserial. Use by calling “import serial”. ©2015 Chris Liechi [clichi@gmx.net](mailto:clichi@gmx.net) All Rights Reserved.

Usage: <https://pyserial.readthedocs.io/en/latest/pyserial.html>

### crc8\_dallas:

This is a CRC-8 library that uses the exact polynomial we need for this application: x^8 + x^2+x+1. I had to modify the code to work with Python 3, since it was originally developed for Python 2.

### sys:

Comes with every distribution of python. Necessary to have a test bench.

Usage: <https://docs.python.org/3/library/sys.html>

### json:

Comes with every distribution of python. Necessary to convert dictionaries into JSON format and dump it directly to an outfile.

Usage: <https://docs.python.org/3/library/json.html>

# Initialization:

Create a BMS() object, passing in PORT and BAUDRATE. This will initialize the serial connection to the BMS PLC.

The BMS() object will destroy itself when python exits.

# Classes:

## BMSStatistic():

### Description:

An internal class that contains a statistic from the sentence SS1(). Makes it easier to do mass data collection from a series of sentences if a request for every statistic available is made.

### Variables:

Every BMSStatistic object contains at least 4 variables:

* statisticIdentifier: what is the ID of this statistic (i.e. what protocol to use to process it)
* statisticValue: what is the value spat out (in decimal converted earlier from hexadecimal) from the BMS system?
* statisticValueAdditionalInfo: any additional information spat out from the BMS system (e.g. Cell ID)?
* timestamp: what time (in seconds since January 1, 1970 at 00:00 GMT) recorded. The BMS system records it in seconds since January 1, 2000 at 00:00 GMT).

Possible additional variables the class can have:

* Name: What is the real name of the statistic?
* Unit: what unit is the value recorded in (e.g. V, mA, W)? If N/A, the value is simply how many times an event occurred.
* Cell\_ID: What is the ID of the cell the statistic came from?

### Methods:

* .dict(): converts this class into a dictionary with keys being the class variables it has, and their corresponding values.
* .string(): converts this class into a string in JSON format.
* .\_\_init\_\_(): initializes the object. Takes statisticIdentifier,statisticValue,statisticValueAdditionalInfo,timestamp. Upon creation, runs a specific protocol to process the data based on its statisticIdentifier.

## BMS():

### Description:

A class that can read the BMS system. Call the .DumpToJSONFile() method to dump all data to an outfile. Details below.

### Variables:

Every BMS() object contains at least 3 variables:

* PORT: what port number is the Raspberry Pi reading from?
* BAUDRATE: at what baudrate (in bits/second) is the Raspberry Pi reading at?
* ser: the serial object (from the pyserial library) that sends and receives data from the BMS.

### Methods:

* .VR1(): returns a dictionary containing hardware type, serial number, and firmware version.
* .BB1(): returns a dictionary containing number of cells, minimum balancing rate, and average cell balancing rate.
* .BB2(): returns a dictionary containing cell string number, first cell number, size of group, and individual cell module balancing rate of each cell group.
* .BC1(): returns a dictionary containing battery charge, battery capacity, and state of charge.
* .BT1(): returns a dictionary containing the summary of cell module temperature values of the battery pack.
* .BT2(): This sentence contains individual cell module temperatures of a group of cells. Each group consists of 1 to 8 cells. This sentence is sent only after Control Unit receives a request sentence from external device, where the only data field is ‘?’ symbol. The normal response to BT2 request message, when battery pack is made up of two parallel cell strings:
* .BT3(): This sentence contains the summary of cell temperature values of the battery pack. It is sent periodically with configurable time intervals for active and sleep states (Data Transmission to Display Period).
* .BT4(): This sentence contains individual cell temperatures of a group of cells. Each group consists of 1 to 8 cells.
* .BV1(): Returns a dictionary containing a summary of cell voltages. contains number of cells, minimum cell voltage, maximum cell voltage, average cell voltage, and total voltage.
* .BV2(): This sentence contains individual voltages of a group of cells. Each group consists of 1 to 8 cells.
* .CF2(parameterID): returns the parameter data of the parameter ID. Must be processed separately.
* .CG1(): This sentence contains the statuses of Emus internal CAN peripherals. Can include CAN current sensor, and CAN cell group, along with the cell group number.
* .CN1(): This sentence reports the CAN messages received on CAN bus by Emus BMS Control Unit, if “Send to RS232/USB” function is enabled.
* .CN2(): This sentence reports the CAN messages sent on CAN bus if "Send to RS232/USB function is enabled.
* .CS1(): Returns a dictionary containing the parameters and status of the charger. Includes set voltage, set current, actual voltage, actual current, number of connected charger, and CAN charger status.
* .CV1(): Returns a dictionary containing the values of total voltage of battery pack, and current flowing through the battery pack.
* .DT1(): This is a placeholder for an electric vehicle sentence. The code is being specifically programmed for a greenhouse, so this sentence will not be programmed and return an error.
* .FD1(): This sentence resets the unit to factory defaults. Use at your own risk.
* .IN1(): This sentence returns a dictionary containing the status of the input pins (AC sense, IGN In, FAST\_CHG).
* .LG1(clear): This sentence can either: retrieve events logged, or clear the event logger.
  + Retrieve Events Logged: pass in ‘N’ or a null value.
    - Every event is recorded in a dictionary form like this: [“log event number 1”]: [“log event”: “No event”, “unix time stamp”: 1567014467
    - Possible events:
      * No Event
      * BMS started
      * Lost communication to cells
      * Established communication to cells
      * Cells voltage critically low
      * Critical low voltage recovered
      * Cells voltage critically high
      * Critical high voltage recovered
      * Discharge current critically high
      * Discharge critical high current recovered
      * Charge current critically high
      * Charge critical high current recovered
      * Cell module temperature critically high
      * Critical high cell module temperature recovered
      * Leakage detected
      * Leakage recovered
      * Warning: low voltage – reducing power
      * Power reduction due to low voltage recovered
      * Warning: high current – reducing power
      * Power reduction due to high current recovered
      * Warning: High Cell module temperature – reducing power
      * Power reduction due to high cell module temperature recovered.
      * Charger connected
      * Charger disconnected
      * Started pre-heating stage
      * Started pre-charging stage
      * Started main charging stage
      * Started balancing stage
      * Charging finished
      * Charging error occurred
      * Retrying charging
      * Restarting charging
      * Cell Temperature Critically high
      * Critically high cell temperature recovered
      * Warning: High cell temperature – reducing power
    - Unix Timestamp: Time recorded in seconds since January 1, 1970 at 00:00 GMT.
    - Log event number: what event number it
  + Clear Event Logger: pass in the ascii value ‘C’ or ‘c’.
* .OT1(): Returns a dictionary containing the status of output pins (Charger pin, heater, bat. low, buzzer, chg. ind.)
* .PW1(request, password): Check the admin status with PW1(‘?’). Log into BMS system with PW1(‘P’, password). Logout with PW1().
* .PW2(request, newPassword): Sets a new password, or clears a password. To set new password, call PW2('S',"mynewpassword"), and substitute “mynewpassword” with whatever password you want. To clear your password, call PW2('C'). Returns true if successful, false if not successful.
* .RC1(): Resets the current sensor reading to zero. Used after current sensor is initially installed.
* .RS1(): Resets the Emus BMS control unit entirely. Like a sudo reboot on a linux machine. Requires admin clearance.
* .RS2(): This sentence is used to retrieve the reset source history log.
* .SC1(percentage): This sentence sets the current state of the charge of the battery in %. Send in an integer from 0 to 100. This method will convert to hexadecimal format first. Returns False if not successful or invalid percentage is passed. Returns True if successful.
* .SS1(request, statisticIdentifier): This sentence can either: Request All Statistics, Request a Specific Statistic (pass in a number), or Clear all unprotected statistics.
  + Request All Statistics: call SS1(‘?’). This will return all statistics the BMS currently has in the form of dictionaries converted from BMSstatistic classes.
  + Request a Specific Statistic: call SS1(‘N’, number), where number is a positive integer. Returns a dictionary containing a single statistic.
  + Clear all unprotected statistics: call SS1(‘c’).
* .ST1(): This sentence returns the status of the BMS in dictionary form. It contains these statistics:
  + **Charging flags**: charging stage, last charging error, last charging error parameter (for debugging purposes), stage duration,
  + **Status flags:** Valid cell voltages, Valid balancing rates, valid number of live cells, battery charging finished, valid cell temperatures
  + **Protection flags**: undervoltage, overvoltage, discharge overcurrent, charge overcurrent, cell module overheat, leakage, no\_cell\_comm, cell\_overheat
  + **Power flags**: warning: power reduction: low voltage, warning: power reduction: high current, warning: power reduction: high cell module temperature, warning: power reduction: high cell temperature
  + **Pin flags**: no\_function, speed\_sensor, fast\_charge\_switch, ign\_key, charger\_mains\_AC\_sense, heater\_enable, sound\_buzzer, battery\_low, charging\_indication, charger\_enable\_output, state\_of\_charge, battery\_contactor, battery\_fan, current\_sensor, leakage\_sensor, power\_reduction, charging\_interlock, analog\_charger\_control, ZVU\_boost\_charge, ZVU\_slow\_charge, ZVU\_buffer\_mode, BMS\_failure, equalization\_enable, DCDC\_control, ESM\_rectifier\_current\_limit, contactor\_precharge
* .TD1(): Returns time and date according the BMS in dictionary form. Returns year, month, day, hour, minute, second, and the amount of uptime the unit has in seconds.
* .TC2(): Used to calibrate cell temperature by a PC, not a microcontroller.
* .DumpToJSONfile(outfile): Calls all data methods listed above, then dumps all data returned into an outfile in JSON format.

Note: Every data harvesting method returns “Cannot communicate to cells” if it fails.

# Private Methods:

bitAt(bitfield, position):

### Description:

Returns True if the bit is 1 at the position of the bitfield, False of 0. Used to analyze bitfields with fewer lines.