BMS.py Design Document

Author: William Kerr

Table of Contents

[Introduction: 2](#_Toc17884669)

[Dependencies: 2](#_Toc17884670)

[pyserial: 2](#_Toc17884671)

[crc8\_dallas: 2](#_Toc17884672)

[sys: 2](#_Toc17884673)

[json: 2](#_Toc17884674)

[Initialization: 2](#_Toc17884675)

[Classes: 2](#_Toc17884676)

[BMSStatistic(): 2](#_Toc17884677)

[Description: 2](#_Toc17884678)

[Variables: 2](#_Toc17884679)

[Methods: 3](#_Toc17884680)

[BMS(): 3](#_Toc17884681)

[Description: 3](#_Toc17884682)

[Variables: 3](#_Toc17884683)

[Methods: 3](#_Toc17884684)

[Private Methods: 6](#_Toc17884685)

[bitAt(bitfield, position): 6](#_Toc17884686)

[Description: 6](#_Toc17884687)

# 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.