Documentation for Embedded Hardware Development

Contents

[**System Requirements**](#_k7mdbe2l1v1p) **3**

[System Capabilities](#_7s99pawlqcxb) 3

[System Behavior](#_zfwtwe1gl11n) 3

[EV Device vs BSS Device](#_uk1f2htrsstu) 4

[**Hardware Level Implementation**](#_dpdshp2rpwqd) **5**

[Components and Connections](#_ycrec8gah7ao) 5

[**Software Level Implementation**](#_j6mdvd969gp4) **7**

[main.cpp](#_bhdy2h3mufqv) 7

[ESPWifi.h](#_m7h5ldxdjrau) 9

[bluetooth.h](#_eey9g4w56yot) 10

[Storage.h](#_vq7r2irs7d5e) 11

[can.h](#_6gu4ot66jbeh) 12

[rtc.h](#_d7kv81qq6lu) 14

[cmdlib-master.h](#_flt5q144ion9) 14

[ESP32Time.h](#_3zzghouowi28) 15

[ciotc\_config.h](#_lrv790mdpt5k) 16

[defines.h](#_o191c795kaac) 16

[esp32-mqtt.h](#_hzp6e986obi3) 16

# System Requirements

## System Capabilities

* Retrieval of absolute time at any arbitrary instance
* Data reads from CAN bus at 100 ms intervals (currently 2 second)
* Adequate processing capability to perform derivatives and integrals on data received from CAN
* WiFi connection capability
  + Saved access points which can be reconnected with automatically
  + Update of WiFi credentials at runtime
* Transmission of data to the cloud platform
  + Currently the delay between transitions is approximately 1 second
* Storage of data to SD Card
  + Synced with data acquisition
  + If WiFi is available, send stored data to Cloud
* Transmission of Real Time data to phone via bluetooth
  + Synced with the rate of data acquisition from CAN (currently 2 seconds)
* Authentication of bluetooth device
  + Reconnection with previously authorised device only
* Visual indicators for status of various aspects (WiFi, Bluetooth, CAN, power operation etc.)
* Time period of data transmission of data over bluetooth variable at runtime.
* OTA (over the air) firmware update
* Alarm in certain conditions in the form of audiovisual indications
* Admin access over the system capabilities will include? (do we still add this)
  + Getting firmware version
  + Getting data stored in SD card
  + Must be wireless
* Compactness
* Waterproof
* Good protection from electrical interference

## System Behavior

* The device captures data from the CAN bus
* The data passes through a secondary layer, where multiple operations can be performed on this data.
  + Operations include moving average, derivative, integral etc.
* The secondary layer is linked to the tertiary layer, where data is transmitted
  + This layer is also responsible for receiving instructions via bluetooth
  + Bluetooth connection is authenticated and sustained throughout operation whereas there is an allowance for intermittent WiFi connection up to a certain threshold dependant on the storage capacity of the SD card from the parallel layer (see below)
  + Data transmission is carried out via bluetooth and WiFi
* A parallel layer to the tertiary stores data on an SD card

## EV Device vs BSS Device

Essentially both devices have the same libraries and functionality. The key difference lies in the data being sent. Both send differing data from a varying number of batteries (4 batteries for the EV and 16 batteries for the BSS).

Another key difference lies in the individual functionality of these devices in the context of the swapping function. The exact functionality has not been decided yet but the key functions of the EV side are to authenticate itself as a verified EV device to the BSS side and also take note of the values of the battery levels before removal and after replacement. The BSS device must provide the right battery given the battery level of the battery coming from the EV side.

# 

# Hardware Level Implementation

## Components and Connections

The components used to make the embedded device consist of the following:

* ESP32 DEVKIT with ESP32-WROOM-32
* MCO2515 CAN Module
* DS3231 RTC Module
* SD Card Reader

**Note**: Another component previously in use was the HM-10 BLE module however that is not utilized in the current iteration. A shift was made towards the NimBLE library as it makes use of the BLE device that comes with the ESP-32, hence reducing the footprint of our device. It is also relatively lighter than the standard BLE library, allowing for further space optimization.  
**Note**: We were previously using a 2 microcontroller setup in order to handle the multiple modules and the data flow. This resulted in a different pin configuration from the one shown below. The current iteration, however, uses only one microcontroller. This was achieved through the optimization of certain libraries and removal of rudimentary parts of the code.

The connections are as follows:

* RTC
  + RTC\_VCC -> ESP\_3V3
  + RTC\_GND -> ESP\_GND
  + RTC\_SDA -> ESP\_D21
  + RTC\_SCL -> ESP\_D22
* SD Reader
  + SD\_VCC -> ESP\_3V3
  + SD\_GND -> ESP\_GND
  + SD\_CS -> ESP\_D5
  + SD\_SCK -> ESP\_D18
  + SD\_MISO -> ESP\_D19
  + SD\_MOSI -> ESP\_D23
* CAN
  + CAN\_VCC -> ESP\_3V3
  + CAN\_GND -> ESP\_GND
  + CAN\_TX -> ESP\_RX2
  + CAN\_RX -> ESP\_TX2

Todo: recheck these connections tomorrow.

There are two devices made for the BSS and EV. Their key differences are explained below.

Provided below is a hardware level flow diagram for the embedded device:

# 

# Software Level Implementation

## main.cpp

This acts as the wrapper function to carry out all of the required tasks through their respective functional pipelines. Task handles are created for all the overarching functions to be scheduled at intervals. The setup first initializes all the devices with their respective init functions and sets certain pins as pin outputs for led indicators. Then, by making use of semaphores and RTOS tasks, all of the overarching functions are initialized. We also made use of the two threads in the esp32 to schedule tasks more efficiently. The list of overarching functions includes:

1. Data Acquisition (vAcquireData)
   1. Acquires data via the CAN bus.
   2. Synchronized with the bluetooth transfer function
   3. Data sent varies according to the device (BSS or EV)
   4. Currently fetches dummy data
   5. Makes use of the RTC library (rtc.h)
2. Bluetooth Transfer (vBlTransfer)
   1. Synchronized with the data acquisition function
   2. Sends data received from the CAN bus via bluetooth
   3. Target device is a phone
   4. Makes use of the bluetooth library
   5. Restricts usage of bluetooth by taking all bluetooth related semaphores (semaBlTx1 and semaBlRx1)
3. Bluetooth Commands (vBlCheck)
   1. Checks for bluetooth commands sent via phone and executes them accordingly
   2. It takes the wifi and bluetooth reception semaphore (semaBlRx1 and semaWifi1)
   3. Note: Any command that takes a resource must take the corresponding semaphore
   4. It makes use of the bluetooth library (bluetooth.h)
4. Storage Handler (vStorage)
   1. Stores data on the SD card in the event that wifi is unavailable
   2. Takes semaphores for storage, data acquisition and wifi (semaStorage1, semaAqData1, semaWifi1)
   3. Makes use of storage and RTC libraries (Storage.h, rtc.h)
5. Transfer data on Wifi (vWifiTransfer) ask AR about certificates
   1. Checks for unsent data on SD card and sends it to cloud
   2. Makes use of MQTT protocol
   3. Takes semaphore for WiFi (semaWifi1)
   4. Makes use of WiFi, MQTT and storage libraries (ESPWifi.h, esp32-mqtt.h, storage.h)
6. Status LED (vStatusLed)
   1. Modifies the LEDs according to the status of peripherals
   2. Makes use of WiFi and bluetooth libraries (ESPWifi.h and bluetooth.h)

Provided below is a diagram for the RTOS functions in main.cpp, mapped according to their priority:



## ESPWifi.h

1. bool init()
   1. This function initializes the ESP wifi module with the requisite configurations.
   2. It attempts to connect to a default SSID and password
   3. It also updates the vector for APs in the WiFiMulti class.
2. void update\_APs()
   1. This function adds new APs to the WiFiMulti class
   2. This should only be called on the first initialization as it may result in multiple instances of the same credentials in the vector
3. bool create\_new\_connection(const char \*SSID, const char \*Password)
   1. This function adds a new connection to the vector in the WiFiMulti class and the text file for APs in the Sd card
   2. In the situation that the SSID sent is the same as one that already exists, the function will either fail if the password is the same or update the specific entry if not.
   3. “SSID” and “Password”refer the respective parameters of the new AP
4. bool remake\_access\_points()
   1. This function serves to destroy and recreate a WiFiMulti class in order to change the list of APs
   2. This is done as the AP list (vector) is a private variable
5. bool connect\_to\_nearest()
   1. This function is used to connect to the nearest available access point
6. bool check\_connection()
   1. This function is a wrapper function which checks the status of the wifi connection.
   2. It is designed to be used in a loop
   3. Can be altered according to requirements
   4. Returns true if a previous connection exists, false otherwise

## bluetooth.h

1. The classes for service callbacks, characteristic callbacks, and descriptor callbacks are defined by default in the NimBLE library but they allow for additional functionality and responsiveness to our code as they allow for the use of callback functions
   1. Currently we’re using these callbacks to change the status of two status flags
   2. It is advisable to not use complex logic in these callbacks as they could potentially disrupt and defer the RTOS tasks.
      1. Some more critical tasks may become obsolete if deferred for too long.
2. bool init()
   1. This function initializes the bluetooth
   2. The name of bluetooth is defined in macro BLUETOOTH\_NAME present in header file
      1. This initializes the name of the server and the security mode for the bluetooth server
   3. It additionally defines all the requisite services and characteristics, with the required properties and security levels
   4. Additionally callbacks are associated to the respective components in this function as well
3. bool send(String tosend)
   1. Sends string of response data on bluetooth upon receiving instructions
   2. This function is separate from the function that sends CAN data to the phone and deals singularly with responses to instructions given from the phone
   3. “tosend” refers to the data to be sent
4. bool send\_notification(String tosend)
   1. this function deals with sending the CAN data in the form of periodic notifications to the phone
   2. “tosend” refers to the data to be sent
5. void display(String ID, String Username, String Password)
   1. This is a simple display function for diagnostic purposes
6. bool check\_bluetooth()
   1. This is a wrapper function which checks for incoming bluetooth messages and responds to them accordingly
   2. It uses some other functions defined previously in this library

## Storage.h

1. bool init\_storage()
   1. Initializes storage
2. bool write\_data(String timenow, String data)
   1. “timenow” is the current time and “data” is the data to be sent
   2. Data is encapsulated in “<>” to separate it
   3. It first checks for free space more than a certain threshold. Otherwise it deletes the oldest data
   4. It also checks for mount success and file presence
   5. It creates a config.txt file on the first ever write
3. bool write\_AP(String SSID, String Password)
   1. “SSID” and “Password” refer to the respective parameters of the AP (Access Point)
   2. Checks for mount success and file presence
   3. Adds an access point to the file APs.txt
4. bool rewrite\_storage\_APs(String SSID[10], String Password[10])
   1. This function clears the APs.txt file and adds new credentials according to the String passed to it. It is called whenever the AP list exceeds a set limit or a saved SSID is being connected to with a different password.
   2. SSID and Password refer to the lists to add
5. void create\_header(File file)
   1. Adds header to the given file
6. void remove\_oldest\_file()
   1. Loops over all files to check the oldest timestamp and deletes that file
7. String read\_data()
   1. Returns a string of the data without the encapsulation “<>”
   2. Updates the size of the current chunk
8. return\_APList()
   1. This function cycles through the APs stored in the SD card and stores them in a String array.
   2. Since the String array parameter degenerates to a pointer it changes the original referenced parameter and thus returns the list of credentials in two arrays.
9. void mark\_data()
   1. timenow refers to the time of the data to be read
   2. Updates the current read pos in config.txt
   3. If the remaining data in file is less than 10 it also updates the filename
10. String next\_file(String curr\_file)
    1. “curr\_file” refers to the current file in use
    2. Returns a string for the name of the next file
11. long get\_unsent\_data(String timenow)
    1. Returns unsent data in bytes
    2. timenow refers to the time of the current file

## can.h

1. EvData is a struct that refers to the electric vehicle data recieved via the CAN bus
2. BmsData is a struct for a single battery cradle unit in the BSS. The data in it is received via the CAN bus
3. bool init\_can()
   1. Initializes CAN module
   2. The speed of CAN and other parameters are decided by the standard kept by Koreans
   3. The intialization is tried 5 times
4. void send\_msg(uint16\_t id, float soc, float hi\_temp, float lo\_temp, float voltage, float current)
   1. Sends message on CAN bus
   2. It is only for emulation of CAN protocol developed by the Koreans
   3. It is only used in hardware for emulation
   4. This function sends data on CAN bus according to standard defined by Koreans
   5. “id” is the 11-bit ID in hex, “soc” is the state of charge, “hi\_temp” is the BMS high temperature, “lo\_temp” is the BMS low temperature, “voltage” and “current” refer to the BMS voltage and current respectively
5. bool receive\_msg()
   1. Reads data from CAN bus
   2. First, it checks if data is available to be read. If there is data available
   3. Then it reads data according to prestored ids as provided by Koreans
   4. It does not return the data rather it updates the variables in the class object itself.
6. void mcu\_message()
   1. Sets the values of the variables in the evdata struct using the data received via the CAN bus
7. void ucycle\_message()
   1. sets the utilized cycles of the batteries in the BSS via the CAN data received
   2. “id” refers to the battery slot and “data” is the CAN data received
8. void cvts\_message()
   1. sets the current, voltage, state of charge and temperature (CVTS) of each battery using the CAN data received
   2. “id” refers to the battery slot and “data” refers to the CAN data received
9. void soh\_message()
   1. Sets the state of health of each battery in the BSS using the CAN data received
   2. “id” refers to the battery slot and “data” is the CAN data received
10. void bid\_message()
    1. updates the battery ID in each battery slot in the BSS according to the data received from the CAN bus
    2. “id” refers to the battery slot and “data” is the CAN data received

## rtc.h

1. bool initRTC()
   1. Initializes RTC and adjusts date and time in case of power loss
2. String getTime()
   1. Function getTime returns time from RTC hardware in form of string
   2. The format is : "YYYY-MM-DD HH : MM : SS"
3. String getTime2()
   1. Function getTime2 returns time from RTC hardware in form of string
   2. The format is: "YYYYMMDD"
   3. Difference from getTime() function is the format
   4. It is required for writing data in file
4. String getNextDay(int iyear, int imonth, int iday)
   1. Returns the next day, given the current time
5. String unixTime()
   1. Converts the current time into unix format
6. void \_set\_esp\_time()
   1. Sets the current time of the ESP32 using the RTC hardware

## cmdlib-master.h

1. String parse\_by\_key(String message, int key)
   1. This is a generalized function used to parse the message (after the ID) to return a value at a certain position (or key) in the message
   2. “message” refers to the received message in String format, and “key” refers to the data entry to extract
      1. The key must never be larger than the total number of entries in the message
   3. The function returns a string of the data in the frame picked using the key
2. String command\_3\_newConn(String message)
   1. This command is used to create a new connection for WiFi and store it in the list of APs
   2. “message” refers to the string containing the new SSID and Password to connect to
   3. Returns an empty string on success or an error string otherwise
3. String command\_5\_enterSwap()
   1. This command initiates battery swap mode and saves the initial cycles for subtraction when the battery swap mode is exited
   2. Returns an empty string on success or an error string otherwise
   3. The data is simulated as we aren’t receiving data from the CAN bus
4. String command\_6\_exitSwap()
   1. This command exits battery swap mode and takes the final cycles of the swapped batteries and sends the difference via bluetooth
   2. Returns an empty string on success or an error string otherwise
      1. This function is not sending any data back as we still aren’t receiving any data from CAN
5. String command\_7\_checkWifi()
   1. This command tells the slave to check the wifi connection and send the status back to the master
   2. Returns “connected” if connection is established or “disconnected” otherwise
6. String command\_8\_getTime()
   1. Returns the unix time as a String
7. String command\_bt(string message)
   1. This is the main wrapper function that is called in a loop and checks for commands on bluetooth
   2. “message” refers to the command sent from the phone
   3. It returns the response to the instruction using the corresponding ID of the message and calling the respective command

## ESP32Time.h

This library is picked up as is from the internet. It is essentially used to set the time of the ESP. All functions have documentation and are rather self explanatory.

## ciotc\_config.h

This file contains your configuration used to connect to Cloud IoT Core. It is picked and used as is.

## defines.h

This file contains some globally used variables for the embedded system.

## esp32-mqtt.h

This file contains the necessary functions to send data to the cloud using the mqtt protocol. This library is picked and used as is.

Power rating

Temperature Range

Any known issues