UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

GESTIONE DELLA BATTERIA DI ALTA TENSIONE NELLE MACCHINE ELETTRICHE

Relatore: Prof. Mauro Migliardi

Laureando: Xhemali Sheshori

ANNO ACCADEMICO 2021 – 2022

Data di laurea 17 marzo 2022

Indice

1 Introduzione

1.1 Formula SAE …….3

1.2 RaceUp Team…….3

1.3 Regolamento Formula Student…….……..4

1.4 Batteria in un'auto elettrica…….……...4

2 Strumenti Utilizzati

2.1 Arduino Due……..…….……...7

2.2 BMS Slave……..…….…….....8

2.3 SFP200MOD……..…….……...9

2.4 Protocollo Controller Area Network (CAN) …….9

3 Progettazione

3.1 Premessa…….……..…….....11

3.2 Protocollo di comunicazione Host-Slave……..…….....11

3.3 Gestione BMS Host……...……...14

3.4 Gestione BMS Slave……...……..……15

4 Implementazione

4.1 Software BMS Host……..…….....18

4.2 Software BMS Slave……..…….....19

5 Collaudo

5.1 Test……..……..…….....20

5.2 Conclusioni……..……..….....20

A Codice Host……..22

B Codice Slave…….37

Capitolo 1

Introduzione

1.1 Formula SAE

Formula SAE è una competizione internazionale universitaria di design ingegneristico, nata nel 1981 con lo scopo di dare agli studenti universitari l'opportunità di mettere in pratica quanto appreso durante il proprio percorso di studi. Questa competizione, proposta inizialmente dalla Society of Automotive Engineers (SAE), prevede la progettazione e la produzione di un'auto da corsa che deve essere valutata per le sue qualità di progettazione e di efficienza ingegneristica. Essa consiste in eventi annuali che hanno sede tutto il mondo, organizzati direttamente dalla SAE o dalle varie associazioni nazionali di ingegneri dell'automobile.
La competizione si è evoluta rispetto alla prima edizione dell'81 e prevede tre classi: *Combustion* (Classe 1C) per i veicoli a motore a combustione interna, *Electric* (Classe 1E) per i veicoli elettrici, *Driverless* (Classe 1D) per i veicoli a guida autonoma.
A partire dal 2017 la Formula Student Germany, ha proposto la propria versione della competizione rinominandola "Formula Student", aggiungendo qualche variazione nel regolamento base pensato dalla SAE. L'idea che oggigiorno sta alla base degli eventi organizzati da Formula SAE è quello di un'azienda fittizia che ingaggia il team affinché realizzi un'auto da corsa. Ogni gruppo di studenti riceve una valutazione in termini di presentazione del paino aziendale, analisi dei costi e pubblicazione del veicolo, oltre alle prove di *accelerazione*, *skid-pad*, *autocross* ed *endurance*. Durante il test di accelerazione la vettura deve accelerare per 75 metri lungo un percorso rettilineo su superficie piana. La prova denominata skid-pad valuta la capacità in curva della vettura in quanto il tracciato richiama la figura di un 8. L'autocross, invece, è una prova di sprint per valutare la maneggevolezza della vettura. Infine, la prova di endurance mira a valutare la performance complessive del veicolo, in particolare l'autonomia per le auto *Electric* e l'efficienza per le macchine *Combustion*.
I principali eventi europei sono *Formula ATA* in Italia, *Formula Student East* (FSEAST) in Ungheria, *Formula Student Austria* (FSA) in Austria e *Formula Student Germany* (FSG) in Germania che è l'avvenimento più importante dell'anno per tutti i team che partecipano al progetto.

1.2 RaceUp Team

Nel 2006 viene fondata la squadra dell'Università di Padova, chiamatosi RaceUp Team [1], sotto la supervisione del faculty advisor Prof. Giovanni Meneghetti [2]. La squadra è composta da due divisioni. Una divisione che si occupa della progettazione e realizzazione della macchina *Electric* e una divisione che si occupa della progettazione e realizzazione della macchina *Combustion*. Ogni divisone è formata da più reparti, ognuno dedicato a un ruolo specifico nell'ideazione e completamento finale dell'automobile. Oltre all'Università di Padova che finanzia e sostiene questo progetto, ci sono numerose aziende padovane e internazionali che sponsorizzano mettendo a disposizione materiale, consulenza e spazi dove gli studenti possono lavorare. Gli sponsor [3] principali sono OZ Racing che fornisce i cerchioni per le vetture e mette a disposizione l'officina dello stabilimento di San Martino di Lupari, UNOX che fornisce materiale elettrico, consulenza sulla progettazione delle schede elettriche e mette a disposizione un'area di lavoro dedicata ai reparti che lavorano nella parte elettronica delle auto. Infine, Aruba, che mette a

disposizione un private cloud in cui è possibile compiere diverse tipologie di simulazioni che richiedono una capacità di calcolo non raggiungibile dai normali computer.

1.3 Regolamento Formula Student

Gli eventi ai quali partecipa RaceUp Team adottano tutti il regolamento adottato da Formula Student Germany. Il regolamento [4] impone dei limiti che devono essere rispettati durante la progettazione e realizzazione di ogni componente dell'auto. La sezione EV 5.8 indica le linee guida da seguire per la gestione della batteria, BMS (Battery Management System), di alta tensione tra qui le principali sono:

- Monitorare la tensione di tutte le celle all'interno del pacco batteria
- Monitorare la temperatura almeno del 30% delle celle, in modo uniforme all'interno del pacco batteria. Il punto di misura deve essere a non più di 10 mm dal percorso di alta corrente
- Spegnere la vettura se vengono superate le soglie di sicurezza per un tempo superiore a 500ms
- Spegnere la vettura se la temperatura di 60°C viene superata per un tempo maggiore di 1s

Per soddisfare queste richieste sono stati progettati e realizzati dispositivi specifici che servono a monitorare le tensioni e le temperature delle celle. In caso di errori bisogna assolutamente spegnere la vettura per rendere sicura l'incolumità del pilota.

1.4 Batteria in un'auto elettrica

Una batteria per veicoli elettrici è la fonte di energia utilizzata per alimentare la propulsione dei motori elettrici. L'intero pacco batteria (vedi Fig. 1.1) usato nel prototipo Electric, creato da RaceUp Team, è realizzato all'interno dei reparti. Questa realizzazione prevede collaborazione di più reparti; il reparto *Monocoque & Composites* che si occupa della progettazione e della realizzazione del case che conterrà; le celle della batteria, le schede per il monitoraggio delle celle e il cablaggio, il reparto *Electronics* che si occupa della progettazione delle schede di monitoraggio e l'implementazione del cablaggio, il reparto *Power Electronics* che si occupa dell'assemblaggio delle celle, il reparto *Cooling* che fornisce un sistema di raffreddamento e infine il reparto *Software,* di cui faccio parte anch'io*,* che ha il compito di progettare e implementare il firmware delle schede di monitoraggio e di usare i dati acquisiti da queste schede per permettere e rendere efficiente il funzionamento dell'automobile.

Figura 1.1: Render pacco batteria

La batteria realizzata da RaceUp Team è formata da una serie di otto segmenti (vedi Fig. 1.2), ciascuno composto da diciotto serie di due celle in parallelo.

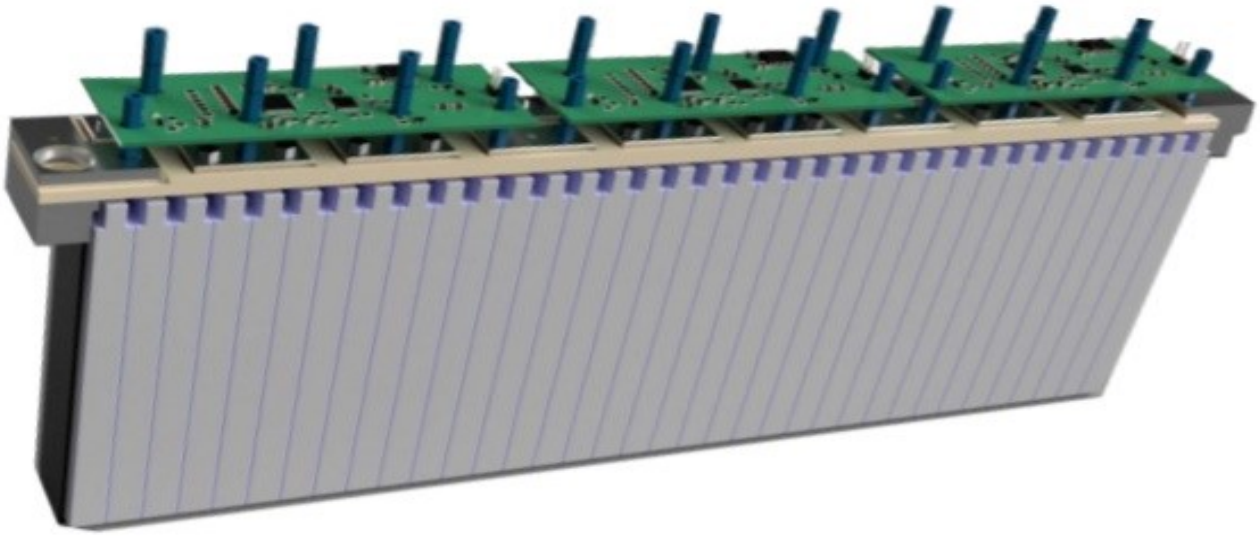

Figura 1.2: Render Segmento

Per la realizzazione dei segmenti sono state scelte le celle Melasta SLPBA843126 [5] (vedi Fig. 1.3) le quali sono ai polimeri di Litio e il formato usato è di tipo Pouch. Sotto sono elencate le principali caratteristiche di una cella che impongono i limiti per la gestione degli errori:

- Tensione massima: 4.2V ± 0.03V
- Tensione di cut off: 3.0V
- Range temperatura in carica: 0°C ~ 45°C
- Range temperatura in scarica: -20°C ~ 60°C

Figure 1.3: Cella Pouch Melasta SLPBA843126

Per rendere possibile il monitoraggio delle celle, sono state sviluppate due tipologie di schede elettriche; BMS host e BMS slave. I moduli BMS slave sono direttamente collegati alle celle e hanno il compito di misurare la tensione di ogni cella e la temperatura del 30% delle celle, come specificato dal regolamento. Il modulo BMS host ha il compito di raccogliere i dati di tutti i BMS slave tramite il protocollo *Control Area Network* (CAN), gestire gli errori, pilotare la ventola di

raffreddamento, leggere la tensione e la corrente dell'intero pacco batteria dal modulo SFP200MOD e trasferire dati in centralina.

Nell figura sottostante (vedi Fig. 1.4) si trova una rappresentazione schematica delle componenti che permettono la gestione software della batteria.

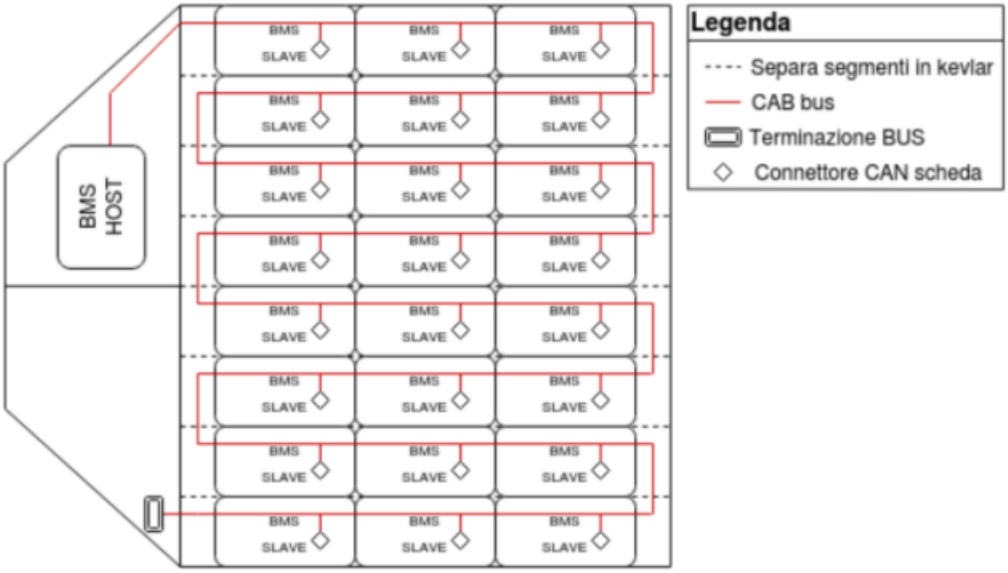

Figura 1.4: Schema pacco batteria

Capitolo 2

Strumenti Utilizzati

2.1 Arduino Due

Per realizzare il BMS host è stato scelto Arduino Due [8], un microcontrollore basato sulla CPU Atmel SAM3X8E ARM Cortex-M3. La scelta è ricaduta su questa scheda in quanto ha un costo basso e dispone di queste caratteristiche:

- Architettura a 32-bit, 84 MHz
- 2 moduli Controller Area Network (CAN)
- 1 modulo Serial Peripheral Interface (SPI)
- Input/Output design a 3.3 V
- 54 Pin *General Purpose Input/Output* (GPIO) programmabili individualmente
- 12 Canali di modulazione di larghezza di impulso (Pulse Width Modulation, PWM)

Per poter rendere usabile questo microcontrollore, il reparto *Electronics* ha progettato una scheda elettrica facilmente collegabile al Arduino Due. Questa scheda integra i transceiver per i moduli CAN e aggiunge un altro modulo CAN, composto dal controller MCP2515 e dal transceiver SN65hvd234, gestito tramite SPI bus.
Le linee della CAN native sono usate per poter comunicare con in BMS Slave e per poter comunicare con la centralina dell'automobile. La linea CAN aggiuntiva invece è stata aggiunto per poter comunicare con il modulo SPF200MOD. Nella Figura 2.1 si mostra il render del BMS host dove la parte in bianco rappresenta lo shield creato dai membri del reparto Electronics e la parte in verde rappresenta l'Arduino Due.

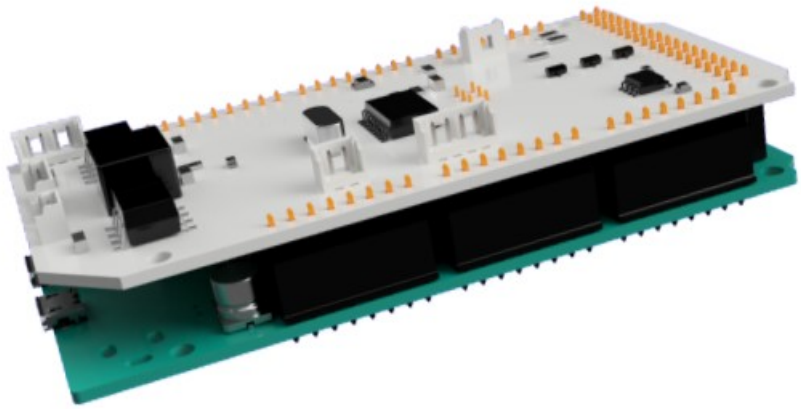

Figura 2.1: Render BMS Host

2.2 BMS Slave

I dispositivi che hanno il ruolo di BMS Slave sono stati progettati dal reparto *Electronics*. Sopra ogni segmento della batteria poggiano tre BMS Slave. Nella Figura 1.2 si vede il posizionamento dei moduli slave nei segmenti. Questi dispositivi hanno il compito di misurare le tensioni e le temperature delle celle e comunicare i dati al BMS host. Inoltre, hanno il compito di mandare al dispositivo host messaggi specifici in caso di malfunzionamento delle celle da esse monitorate. Ogni dispositivo slave misura la tensione di sei celle e la temperatura di due di esse.

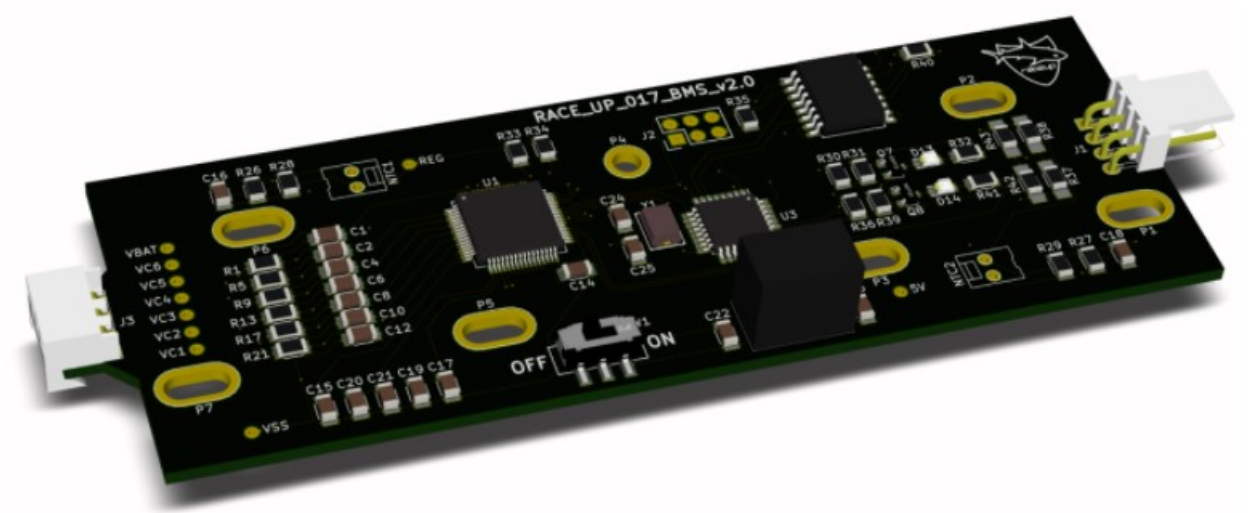

Figura 2.2: Render BMS Slave

Il modulo che si occupa della misurazione delle temperature e delle tensioni è il bq56pl536a-q1 [9]. Questo modulo è in grado di misurare fino a sei tensioni e fino a due temperature, le quali possono essere lette da un microcontrollore esterno tramite un *Serial Peripheral Interface bus* (SPI Bus). Inoltre, dispone di due pin di output che possono segnalare gli errori a livello fisico senza dover interrogare il modulo, un pin di output che segnala che i dati sono pronti per essere letti.

Il microcontrollore scelto per elaborare questi dati è l'Atmega32m1-au [10], scelto oltre al fatto del suo basso costo e basso consumo energetico, anche perché dispone delle seguenti caratteristiche:

- 1 porta seriale Serial Preipheral Interface (SPI)
- 1 modulo Controller Area Network (CAN)
- 27 pin General Purpose Input/Output (GPIO) programmabili individualmente
- 1 modulo Universal Asynchronous Receiver-Transmitter (UART)

La porta seriale SPI serve per comunicare con l'integrato bq56pl536a-q1, il modulo CAN per trasferire i dati al BMS host, i GPIO per pilotare i led di debug e infine il modulo UART per poter rendere più semplice il debuging.

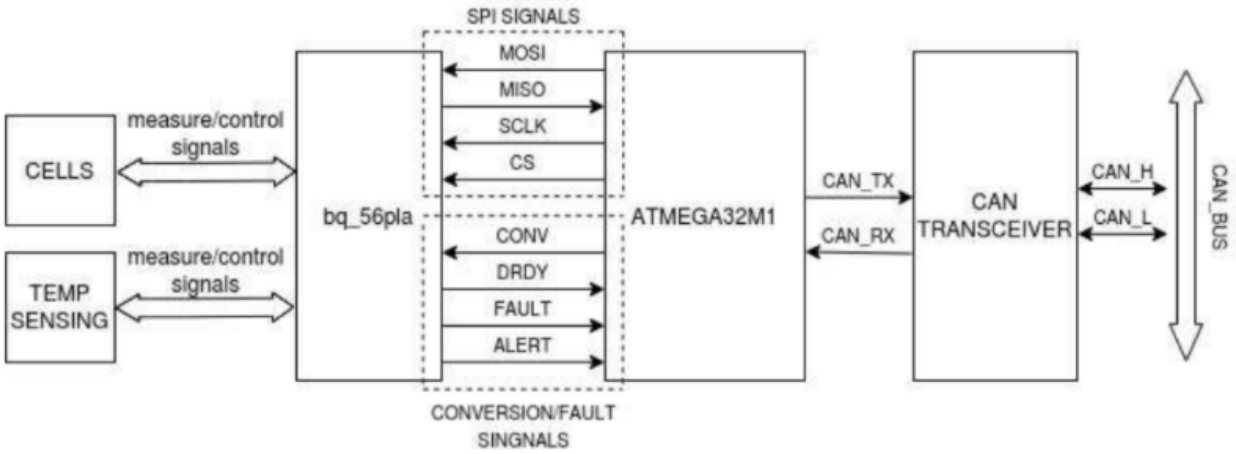

Figura 2.3 Schema a blocchi BMS Slave

2.3 SFP200MOD

Il sensore SFP200MOD *Precision Current and Voltage Measurement Module* [6] è un modulo preconfigurato in grado di leggere contemporaneamente correnti fino all'ordine del Chiloampere e tensioni fino a 1000 Volt. Viene prodotto da Sendyne [7], sponsor ufficiale del RaceUp Team, ed è progettato specificatamente per l'utilizzo in vetture elettriche. Questo sensore presenta un'interfaccia CAN e opera in un range di temperatura da -40 °C a 125 °C con un accuratezza di ± 1.0%. Nella vettura creata da RaceUp Team il sensore SFP200MOD è utilizzato per leggere corrente e tensione della batteria in tempo reale, in modo da poter ricavare la potenza istantanea.

Figura 2.3 SPF200MOD

2.4 Protocollo Controller Area Network (CAN)

Il Controller Area Network è uno standard seriale di comunicazione digitale. Inizialmente introdotto dalla Bosch negli anni '80 per consentire la comunicazione fra i dispositivi elettronici intelligenti montati su un veicolo, oggi è molto diffuso nelle applicazioni industriali di tipo embedded. Le caratteristiche del CAN Bus che lo rendono ideale per questo tipo di applicazione sono:

- Alta immunità ai disturbi elettromagnetici

- Elevata affidabilità: la rilevazione degli errori e la richiesta di ritrasmissione viene gestita autonomamente dall'hardware

- Capacità "Multi-Master": ovvero tutti i nodi della rete possono trasmettere. Nel nostro caso abbiamo in totale 25 nodi che trasmettono e ricevono messaggi (24 "BMS slave" e 1 "BMS host")

I messaggi vengono inviati sotto forma di frame. I frame contenenti i dati del nodo trasmittente si chiamano data frame, e sono composti da un identificatore ID (11 Bit) e un campo dati (0 - 8 Byte), oltre a Start-of-Frame, End-of-Frame e altri bit di controllo, come Data-Length-Code DLC (4 bit) che identifica il numero di byte per codificare ciascun dato. I nodi possono filtrare i data frame in base all'identificatore e scegliere se processare o meno il dato. I frame possono essere anche di tipo *extended.* Se si usa questa versione di frame l'identificatore è composto da 29 bit.

Capitolo 3

Progettazione

3.1 Premessa

Come già detto nelle sezioni precedenti, nella progettazione e nella realizzazione del pacco batteria contribuiscono diversi reparti e di conseguenza molte persone. Facendo parte del reparto Software, il mio compito è stato quello di progettare il protocollo di comunicazione tra host e slave, progettare e realizzare il firmware del dispositivo host, realizzare il firmware dei dispositivi slave e infine quello di testare l'intero sistema software. La progettazione del firmware del dispositivo slave è stata fatta dai membri del reparto Electronics in quanto richiede una conoscenza approfondita del hardware delle schede elettroniche e della fisica delle batterie.

3.2 Protocollo di comunicazione Host-Slave

La batteria della macchina *Electric* progettata da RaceUp Team può operare in quattro modalità diverse ed è compito del dispositivo BMS Host interfacciare l'utente che gestisce la batteria con il sistema. Le quattro modalità sono le seguenti:

- *Normal Mode:* modalità usata quando la macchina è pronta per muoversi o quando si sta muovendo
- *Sleep Mode:* modalità usata quando la macchina è spenta.
- *Printing/Debug Mode:* modalità usata quando si vogliono visualizzare i dati ad un'interfaccia utente.
- *Balancing Mode:* modalità usata quando si sta caricando la batteria

Le modalità sopracitate verranno spiegate più in dettaglio nei paragrafi a seguire.
Tuttavia, la comunicazione tra BMS Host e BMS Slave è gestita tramite quattro tipologie di pacchetti:

- Pacchetti di errore: Error
- Pacchetti Standard: Standard
- Pacchetti di richiesta: Request Mode
- Pacchetti di conferma: ACK

Il fatto che i frame con ID più basso hanno una priorità più alta nel CAN Bus, impone di usare ID bassi per i pacchetti considerati più importanti.
Indubbiamente, i pacchetti più importanti di questo sistema sono i pacchetti di errore. Se un BMS slave manda un messaggio di errore, il BMS Host deve comunicare l'errore in centralina e

immediatamente spegnere la macchina. Questi pacchetti hanno come sorgente i BMS slave e come destinatario il BMS Host. L'ID (vedi *Tab 1.1*) del pacchetto d'errore contiene nei primi 5 bit il numero del BMS slave che ha rilevato l'errore. Invece i restanti 6 bit sono posti a zero. Il payload (vedi *Tab 1.2*) consiste di un solo byte, nel quale i primi 6 bit sono una bitmap che indica le celle che hanno scatenato l'errore. Il settimo bit è posto a 1 se l'errore riguarda la tensione della cella oppure a 0 se riguarda la temperatura. L'ottavo bit è posto a 1 se il dato errato è minore del limite inferiore del range di sicurezza oppure a 0 se è maggiore del limite superiore.

ID											
N. bit	10	9	8	7	6	5	4	3	2	1	0
Funzione	0	0	0	0	0	0	ID BMS SLAVE				

Tabella 1.1 ID Pacchetto d'errore

Payload							
N. bit	7	6	5	4	3	2	1	0
Funzione	0→Over 1→Under	1→Volt 0→Temp	Cell 5	Cell 4	Cell 3	Cell 2	Cell 1	Cell 0

Tabella 1.2 Payload Pacchetto d'errore

I pacchetti standard, invece, sono di due tipi. Con questo pacchetto l'host indica agli slave di preparare i nuovi dati oppure chiede ad essi di mandare i dati che hanno già in memoria. Con la stessa tipologia di pacchetto lo slave manda i dati ottenuti dal bq56pl536a -q1 al host. Il payload del pacchetto che parte dall'host con destinazione gli slave è nullo in quanto l'informazione necessaria è codificata all'interno dell'ID (vedi Tab 1.3). Nei primi 5 bit dell'ID viene specificato il numero del BMS slave di destinazione. Se questi bit sono posti tutti a 1 il pacchetto sarà elaborato da tutti i BMS slave. Il quinto e sesto bit sono riservati e attualmente non sono usati. Il settimo bit è posto a 1 se si vogliono preparare/interrogare le temperature oppure è posto a 0 se si vuole accedere alle tensioni. L'ottavo bit è posto a 0 se lo slave deve preparare i dati (Start Conversion), quindi leggere i registri del modulo bq56pl536a-q1. Se invece l'ottavo bit è posto a 1, lo slave di destinazione deve mandare i dati elaborati all'host (Data Request). Il nono e decimo bit sono posti a 1 per ridurre la priorità di questo pacchetto in quanto è un pacchetto ordinario.

ID											
N. bit	10	9	8	7	6	5	4	3	2	1	0
Funzione	1	1	0→Start Conversion 1→ Data Request	0→ Volt 1→ Temp	-	-	ID BMS SLAVE				

Tabella 1.3 ID Pacchetto standard

Il messaggio mandato dagli slave all'host ha lo stesso formato per quanto riguarda l'ID, ponendo però a 1 il bit numero 8.
Inoltre, allega nel payload i dati richiesti dall'host. Se i dati sono le tensioni delle celle, il payload (vedi Tab 1.4) è formato da 60 bit, 10 bit per la tensione di ogni cella letta dal BMS slave. Nel caso in cui i dati sono le temperature, il payload (vedi *Tab 1.5*) e formato da 2 byte, 1 byte per ciascuna temperatura letta dal BMS Slave

Payload Tensioni							
N. bit	63…60	59…50	49…….40	39…….30	29…….20	19…...10	9……0
Funzione	-	Volt 5	Volt 4	Volt 3	Volt 2	Volt 1	Volt 0

Tabella 1.4 Payload Pacchetto standard tensioni slave → host

Payload Temperature		
N. bit	15……...8	7…….0
Funzione	Temp 2	Temp 1

Tabella 1.5 Payload Pacchetto standard temperature slave → host

I pacchetti "Request Mode" sono utilizzati per poter cambiare la modalità della gestione della batteria. Questi pacchetti sono mandati dall'host agli slave, che in base alla modalità ricevuta elaborano i dati e modificano i loro consumi. Il passaggio alla modalità *Sleep* richiede che gli slave mandino un messaggio di conferma della ricezione. Questa conferma deve avvenire perché una volta in Sleep i slave non comunicano più con l'host fino a che non viene ricambiata la modalità d'operazione. I primi 5 bit dell'ID (vedi Tab 1.6) dei pacchetti "Request Mode" sono posti a 1, in quanto questi pacchetti sono sempre destinati a tutti i slave. Nel sesto, settimo e ottavo bit viene specificata la modalità alla quale bisogna operare. Il decimo bit e posto a 0 per differenziare i pacchetti Standard da quelli Request Mode e l'undicesimo è posto a 1 per diminuire la priorità. È stato scelto di diminuire la priorità di questo pacchetto per poter rendere il più veloce possibile la trasmissione dei pacchetti d'errore.

ID											
N. bit	10	9	8	7	6	5	4	3	2	1	0
Funzione	1	0	0	mode			1	1	1	1	1

Tabella 1.6 ID Pacchetti Request Mode

I bit "mode" sono così interpretati:

- 0b000 *Normal Mode*
- 0b010 *Sleep Mode*
- 0b100 *Balancing Mode*
- 0b110 *Printing Mode*

Quando si vuole operare in Balancing Mode (vedi Tab 1.7), l'host deve indicare la tensione che si vuole raggiungere. Questo dato viene inserito nel payload.

Payload		
N. bit	15……...10	9…….0
Funzione	-	Tensione target

Tabella 1.7 Payload Pacchetto Request Mode balancing

Infine, i pacchetti ACK non hanno nessun payload. Nell'ID (vedi *Tab 1.8*) è contenuta tutta l'informazione necessaria del messaggio.

ID											
N. bit	10	9	8	7	6	5	4	3	2	1	0
Funzione	1	0	1	mode			ID BMS SLAVE				

Tabella 1.8 ID Pacchetto ACK

3.3 Gestione BMS Host

Il BMS host ha il compito di coordinare tutti gli slave, leggere i dati da loro prodotti, fornire un'interfaccia con chi deve monitorare o usare la batteria e trasmettere in centralina i dati che devono essere loggati e gestire la ventola di raffreddamento. In più, deve leggere i dati dal SPF200MOD e mandarli in centralina.

L'utente può interfacciarsi al BMS host connettendosi alla porta micro-USB che possiede l'Arduino Due. L'utente è in grado di leggere le tensioni e le temperature forniti da tutti i BMS slave in modo da monitorare attentamente il ciclo di vita delle celle, grazie alla comunicazione in seriale. Oltre alle letture, sempre tramite la seriale, l'utente può inserire comandi per cambiare la modalità d'operazione della batteria.
I dati che l'host manda in centralina sono la tensione massima, la tensione minima e la tensione media di tutte le celle per fare in modo di avere il controllo dei casi limite ed effettuare il logging di questi dati. Si manda in centralina anche la temperatura massima media e minima per lo stesso motivo. Questi dati vengono elaborati poi dal reparto *Power Electronics* al fine di ottimizzare la gestione e la progettazione del pacco batteria.
Per ciascuna modalità l'host ha un comportamento diverso.
Quando l'utente sceglie di usare la batteria in Normal Mode, che è la modalità che si usa quando il veicolo deve correre, l'host ha il compito di leggere le tensioni e le temperature e gestire gli errori generati dagli slave. Quando l'host riceve un errore deve segnalare l'errore in centralina, mandano lo stesso messaggio ricevuto dallo slave, e spegnere immediatamente la machina facendo scattare il circuito di spegnimento tramite un pin GPIO che controlla questo circuito. La sequenza di operazioni che deve eseguire l'host quando è in Normal Mode è:

1. Mandare un pacchetto di Start Conversion per le tensioni in broadcast a tutti gli slave

2. Mandare un pacchetto di Data Request per le tensioni uno ad uno a tutti gli slave e ricevere le loro riposte. Questo scambio di dati avviene in modo sincrono.
    a. Ogni volta che si riceve un pacchetto, viene controllato se i dati in esso contenuto ricadono nel range di sicurezza.
    b. Se lo slave non risponde entro 500 ms si fa chiudere il circuito di spegnimento e si manda un pacchetto di errore in centralina. Il pacchetto d'errore che indica che uno slave non ha risposto in tempo ha i bit del payload tutti posti a 1 (vedi Tab 1.2).

3. Una volta ricevuti tutti i pacchetti si calcola la tensione media e si individua la tensione massima e minima tra tutte le celle. In base alla temperatura massima si imposta la velocità della ventola di raffreddamento.

4. Si ripetono le operazioni 1, 2, 3 con soggetto le temperature.

Quando si trova nella modalità *Printing Mode*, oltre alle operazioni sopracitate, l'host stampa in seriale una stringa che contiene le tensioni di tutte le celle e le temperature rilevate dagli slave. Questa stringa è di formato JSON e viene presa in input da un programma che rende i dati visibili all'utente.
La modalità *Sleep Mode* viene scelta prima che la macchina venga spenta. Quando si trova in *Sleep Mode*, tutti i moduli che vengono alimentati dalle celle (quindi i BMS slave) devono portarsi in uno stato che impone di usare meno energia possibile. Questo viene fatto per non scaricare velocemente le celle a riposo. L'host ha il compito da comunicare agli slave di passare alla modalità Sleep. Questo viene fatto tramite un messaggio in broadcast a tutti gli Slave. Una volta mandato il messaggio in broadcast l'host resta in attesa che tutti gli slave rispondano con il messaggio di conferma. Se qualche slave non passa alla modalità Sleep vuol dire che quello slave è danneggiato oppure si trova in uno stato non sicuro. Viene segnalato l'errore al reparto *Electronics* che, con la massima cautela e rispettando tutte le regole di sicurezza, devono aprire il pacco batteria e controllare fisicamente lo stato delle celle e dello slave che non risponde e capire il motivo dell'errore.
Durante il processo di carica della batteria, non tutte le celle si caricano nello stesso modo. Questo porta ad avere celle a tensioni diverse e rende difficile la gestione della batteria. Per ovviare questo problema si sceglie la modalità Balancing Mode. Come durante la *Normal Mode*, l'host richiede a tutti gli slave le tensioni e le temperature. Una volta richiesti questi dati l'host manda un pacchetto *Request Mode* in broadcast con *Tensione Target (*vedi *Tab1.7)* la tensione minima tra le celle. Sarà poi compito degli slave portare tutte le celle al valore della *Tensione Target*.

Per quanto riguarda la comunicazione con il modulo SPF200MOD, l'host chiede con una frequenza di 100 Hertz la tensione e la corrente che misura l'integrato. Questa interrogazione avviene indipendentemente dalla modalità in cui si trova il sistema.

Infine, la ventola di raffreddamento è pilotata da uno dei canali di modulazione di larghezza di impulso migliorato (PWM). Viene scelto il duty cycle dell'onda quadra in base alla temperatura più alta rilevata dal sistema. Per temperature sotto i 30 °C il duty cycle rimane costante a 10% e per temperature superiori di 55°C il duty cycle equivale a 90%. Per temperature da 30°C a 55°C il duty cycle è calcolato secondo la formula:

$$\text{duty cycle} = 0{,}08\ \text{temp}_{max}^2 - 3.6\ \text{temp}_{max} + 46$$

Durante la fase di balancing invece si sceglie di usare la ventola con un duty cycle del 90% in quanto il calore dissipato dalla scarica delle celle che devono portarsi alla *Tensione target* è molto elevato

3.4 Gestione BMS Slave

Il compito di ogni BMS Slave è quello di misurare le tensioni di tutte le celle sulle quali è montato e le temperature di due di esse. Inoltre, ha il compito di rispondere alle richieste del BMS Host mandando i dati letti è comportandosi in modo differente in base alla modalità che viene posto.
Ogni BMS slave deve avere un ID univoco all'interno del sistema in modo da rendere possibile l'indirizzamento dei pacchetti dell'host. Per rendere possibile questo su ogni slave risiede un Dip-Switch (vedi Fig 3.1). I piedini dello Switch sono collegati a 5 pin del microcontrollore. Il valore logico di ogni pin compone l'ID dello slave espressa in bit. (Es. se pin 1 e 5 posti a valore logico 1

l'id dello slave sarà uguale a 0b10001). L'ID viene impostato a mano configurando i piedini dello switch. Questa operazione viene fatta prima che lo slave venga montato sulle celle.

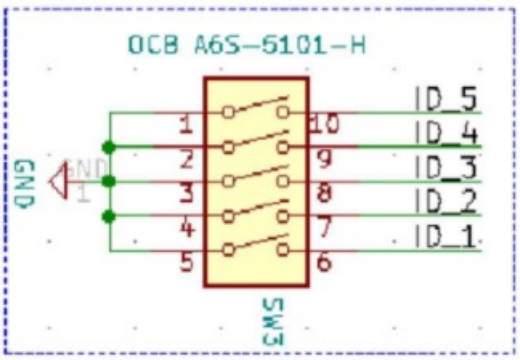

Figure 3.1 Dip-Switch

Le letture delle tensioni e delle temperature sono gestite direttamente dal modulo bq56pl536a-q1. Per poter coordinare le letture, si accedono in scrittura via SPI alcuni suoi registri:

- CONFIG_COV (Over Voltage) 0x42,: Registro che indica la tensione che farà da limite superiore al range di sicurezza delle tensioni. Rispettando le proprietà delle celle usate, questo registro viene posto ad un valore che corrisponde a 4,2 V.

- CONFIG_COVT (Over Voltage Timer) 0x43: Registro che indica il tempo dopo il quale bisogna segnalare un errore di Over Voltage. Quindi se una cella supera il limite superiore del range di sicurezza per più di questo tempo, scatterà l'errore. Nel rispetto del regolamento Formula Student questo registro viene posto ad un valore corrispondente a 500 ms.

- CONFIG_CUV (Under Voltage) 0x44: Registro che indica la tensione che farà da limite inferiore al range di sicurezza delle tensioni. Rispettando le proprietà delle celle usate, questo registro viene posto ad un valore che corrisponde a 3,3 V.

- CONFIG_CUVT (Under Voltage Timer) 0x45: Registro che indica il tempo dopo il quale bisogna segnalare un errore di Under Voltage. Quindi se una cella supera il limite inferiore del range di sicurezza per più di questo tempo, scatterà l'errore. Nel rispetto del regolamento Formula Student questo registro viene posto ad un valore corrispondente a 500 ms.

- CONFIG_OT (Over Temperature) 0x46: Registro che indica la temperatura massima che può raggiungere una cella. Rispettando le proprietà delle celle usate, questo registro viene posto ad un valore che corrisponde a 60 °C.
- CONFIG_OTT (Over Temperature Timer) 0x47: Registro che indica il tempo dopo il quale bisogna segnalare un errore di Over Temperature. Quindi se una cella raggiunge è mantiene la temperatura limite per il tempo indicato in questo registro, scatterà l'errore. Nel rispetto del regolamento Formula Student questo registro viene posto ad un valore corrispondente a 1 s.

- CB_TIME (Cell Balancing Time) 0x33: Registro che indica il tempo dopo il quale deve fermarsi il processo di bilanciamento. Questo dato è estremamente importante in quanto

durante questo processo le celle si scaricano molto velocemente dissipando calore e aumentando in modo significativo la temperatura del pacco batteria.

Invece per leggere effettivamente i dati si accede in lettura ai registri VCELLn, con n che va da 1 a 6, per rilevare le tensioni e ai registri TEMPERATURE1 e TEMPERATURE2 per rilevare le temperature. I registri VCELLn sono registri che contengono dati da 16 bit, però si sceglie di mappare il dato in 10 bit in modo da creare un pacchetto unico per trasferire tutte le misure delle tensioni in una sola trasmissione per rendere più veloce la comunicazione host-slave. Il dato mappato in 10 bit è stato considerato abbastanza preciso dai membri del team che hanno progettato il pacco batteria. L'integrato non può gestire in autonomia le misurazioni, bensì ha bisogno di un dispositivo esterno, che nel nostro caso è il microcontrollore Atmega32m1, che gli indichi quando può effettuarle. Il microcontrollore comunica al bq di cominciare ad effettuare le misure quando riceve un pacchetto Start Conversion in CAN, impostando il pin CONV (vedi FIG 2.3) al valore logico 1. Il bq gestisce tramite interrupt il cambio di stato di questo pin e quando raggiunge un valore logico 1 comincia ad eseguire le misurazioni. Una volta finito il processo di misurazione, che vuol dire che I registri VCELLn, TEMPERATURE1 e TEMPERATURE2 sono stati aggiornati, il bq imposta il pin DRY (vedi FIG 2.3) al valore logico 1. Quando l'Atmega32m1 rivela che il pin DRY ha raggiunto il valore logico 1 effettua la lettura dei registri contenenti i dati, crea pacchetti can che verranno inviati al BMS Host e resta in attesa del pacchetto Data Request. Una volta ricevuto il pacchetto Data Request, lo slave manda il pacchetto richiesto al host.
Oltre a leggere i dati, il bq gestisce anche il bilanciamento del carico delle celle. Quando si riceve il pacchetto Request Mode (vedi TAB 1.6) con i bit che indicano la modalità uguali a 0b100, che corrisponde a Balancing Mode, l'Atmega32m1 effettua queste operazioni:

- Leggere le tensioni di tutte le celle.

- Individuare le celle che hanno una tensione maggiore rispetto alla tensione che l'host ha mandato nel payload e creare una bitmap con bit posto a 1 se la cella ha una tensione superiore. (Es. se le celle che hanno una tensione superiore alla tensione ricevuta dall'host sono le celle numero 0 e 3 la bitmap sarà uguale a 0b100100)

- Scrivere questa bitmap al registro del bq CB_CTRL con indirizzo 0x32. Poi sarà il bq che gestirà la scarica delle celle indicate dalla bitmap per un tempo che equivale al valore contenuto nel registro CB_TIME.

In più, il bq può rilevare eventuali errori riguardanti le tensioni e le temperature. Per quanto riguarda le tensioni il bq confronta i registri VCELLn con i registri CONFIG_CUV e COFIG_COV. Nel caso il valore di VCELLn sia fuori il range di sicurezza il bq imposta il pin FAULT ad un valore logico 1, aggiorna il contenuto del registro FAULT_STATUS, indirizzo 0x21, con un valore che descrive la tipologia di errore. L'Atmega32m1 gestisce il pin FAULT tramite interrupt. Quando questo pin raggiunge il valore logico 1 si legge il contenuto del registro FAULT_STATUS, si manda un pacchetto di errore al BMS Host e poi si resetta l'errore.
Infine, il bq può essere posto in uno stato di Sleep impostando il registro IO_CONTROL a 0b11111100. In questa modalità si riducono al minimo i consumi energetici. Il microcontrollore imposta il bq in modalità Sleep quando riceve un pacchetto di Sleep dall'host.

Capitolo 4

Implementazione

4.1 Software BMS Host

Il software del BMS Host è stato implementato usando Arduino Software (IDE) [11], un'ambiente di sviluppo open-source che può essere usato con un numero molto vasto di microcontrollori.
Il codice è diviso in due parti, setup e loop. Le funzioni di setup vengono eseguite solo una volta all'accensione del microcontrollore e servono per inizializzare i pin GPIO, inizializzare lo stream in seriale per creare l'interfaccia con l'utente, inizializzare le tre linee CAN e le varie mailbox per gestire i diversi tipi di pacchetti. Invece, le funzioni che costituiscono il loop vengono eseguite in continuazione. Queste funzioni sono anch'esse divise in due; le funzioni che gestiscono la lettura della tensione e della corrente dal SPF200MOD e le funzioni che gestiscono l'interfaccia utente e la comunicazione con gli slave e con la centralina. Il BMS Host deve sempre permettere all'utente di interagire tramite linea seriale in modo che l'utente possa cambiare modalità in qualsiasi momento. Nella figura 4.1 è riportato il diagramma di flusso del programma che esegue nel BMS Host.

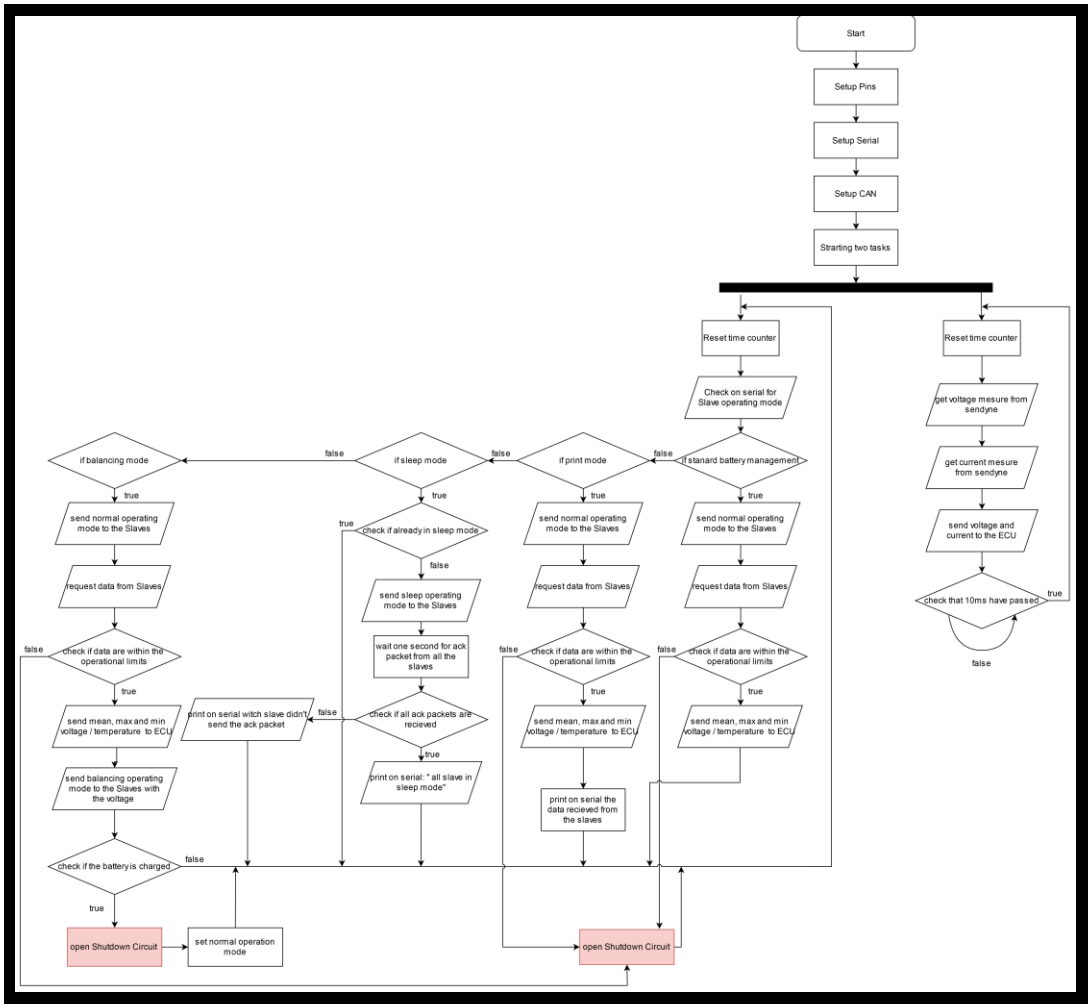

Figura 4.1 Flowchart BMS Host

4.2 Software BMS Slave

Il software del BMS Slave è gestito ad interrupt. Il microcontrollore esegue le operazioni in base ai messaggi che riceve in CAN dal BMS host e in base allo stato del pin DRDY e FAULT del bq56pl536a-q1. Nella parte iniziale del codice si inizializzano i pin GPIO, la connessione in SPI con il bq56pl536a-q1, la linea CAN con le mailbox per gestire le diverse tipologie di pacchetti. Inoltre, viene letto lo stato dello switch per impostare l'ID del BMS Slave. Nella figura 4.2 è riportato il diagramma di flusso del programma che è in esecuzione nell'Atmega32m1.

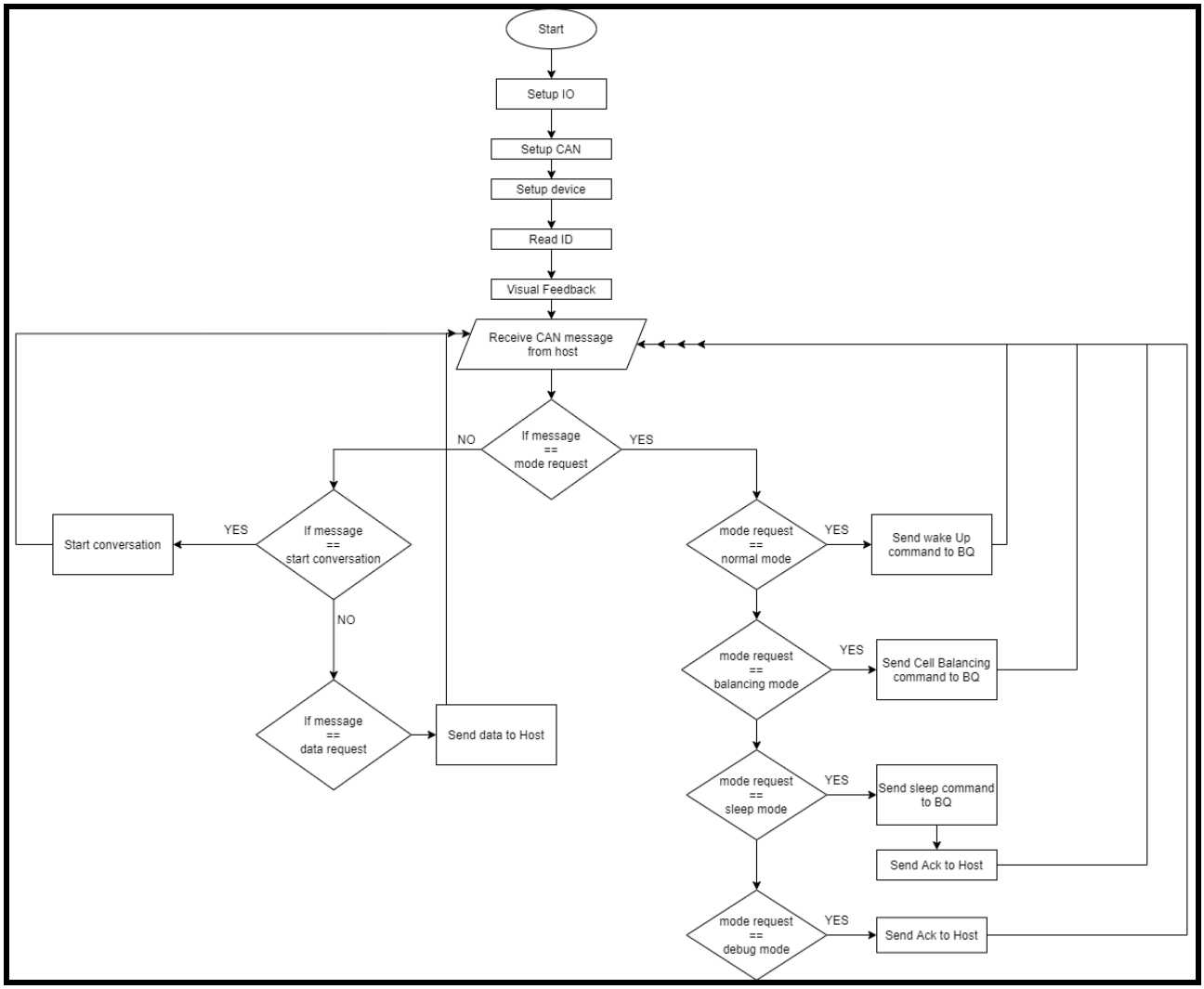

Figura 4.2 Flowchart BMS Slave

Capitolo 5

Collaudo

5.1 Test

Prima di montare il pacco batteria effettivamente in vettura, il regolamento di Formula SAE impone di progettare e realizzare un carrello dove viene collegata elettricamente la batteria, che oltre a permettere lo spostamento in sicurezza dell'intero pacco batteria, permette di effettuare tutti i test di sicurezza e il caricamento della batteria. Formula Student Germany ogni anno pubblica la lista delle operazioni alla quale verrà sottoposto l'automobile durante le ispezioni tecniche che hanno luogo prima delle prove dinamica durante gli eventi. Queste operazioni solitamente si dividono in tre categorie: ispezioni meccaniche, ispezioni dei sistemi elettronici di bassa tensione e le ispezioni del sistema elettronico di alta tensione. Alla fine della realizzazione di ogni componete, I membri di RaceUp team controllano se il loro prodotto rispetta le linee guida di queste ispezioni.
Per quanto riguarda i test sul sistema software della gestione della batteria di alta tensione, è fondamentale che si rilevino gli errori e in caso di presenza di tali si attivi immediatamente il circuito di spegnimento. Per verificare che il sistema rileva gli errori si danno in input agli slave tensioni e temperature fuori dal range di sicurezza e si controlla che l'host faccia scattare il circuito di spegnimento. Per verificare se l'host rileva lo scollegamento di un o slave, si esclude uno slave dalla rete CAN e si controlla se il BMS host spegne la macchina. Si è notato che ogni slave comunica con l'host ogni 226 ms (il limite posto dal regolamento è 500 ms **sezione 1.3**). Inoltre, per vedere le se tensioni lette sono corrette si misurano le tensioni delle celle con un multimetro e si confrontano i valori mostrati dall'interfaccia grafica.
Una volta che i test eseguiti sul carello sono andati a buon fine si passa ai test con la batteria collegata in macchina. La presenza degli inverter aggiunge disturbi elettromagnetici non trascurabili che influenzano nel funzionamento delle schede elettriche e dell'intero sistema. Questi disturbi causano il malfunzionamento dei BMS slave, facendoli rilevare errori che in realtà non sono presenti. Si è notato che schermando gli inverter, i collegamenti della linea CAN e le schede soggette a disturbi, le prestazioni migliorano notevolmente fino a raggiungere un livello di funzionamento accettabile.

5.2 Conclusioni

Durante la presente esperienza nel Race UP team è stata acquisita grande consapevolezza riguardo al sistema della gestione della batteria. Avendo raggiunto un livello di affidabilità accettabile del hardware del sistema, diventa fondamentale migliorare la gestione software per ottimizzare ulteriormente il risparmio energetico cercando soluzioni di gestione differenti in base allo stato di carica della batteria. Salvando i valori prodotti dagli slave, si stanno creando i dati necessari per stimare correttamente la curva di scarica e carica dell'intero pacco batteria.
Per poter ottimizzare la gestione della batteria in un prospetto futuro, diventa estremamente importante la creazione di relazioni dei lavori svolti dove si mettono in evidenza gli obbiettivi raggiunti e gli esperimenti falliti in modo che le conoscenze acquisite durante gli esperimenti vengano trasmesse ai nuovi membri del team.

Appendice A

Codice Host

```
#include "data.h"
#include "setup.h"
#include "serialManager.h"
#include "manager.h"
#include "can_managment_logger.h"
#include "sdc_manager.h"
bool start =true;

void setup()
{
  scaleCellVoltage();
  scaleCellTemp();
  setupPins();
  setupSlaves();
  setupSerial();
  setupInternalCAN();
  setupExternalCAN();
  setupSendyneCAN();
  pwm_pin35.start(PWM_PERIOD_PIN_35, PWM_PERIOD_PIN_35 * 0.1); //Start at 10 percent

  xTaskCreate(SendyneManagment, (const portCHAR *)"SendyneManagment", 128, NULL, 4, NULL);
  vTaskStartScheduler();
}
//Interrogazione SPF200MOD
void SendyneManagment(void *pvParameters)
{
  unsigned long lastTime = millis();

  while (1)
  {
    lastTime = millis();
    if(!start)
    {
      if (receaving_messagies)
        receaving_messagies=false;
      else
        sdcOpen();
    }
    else
      start=false;
    requestSendyneData();

    if (millis() - lastTime <= SENDYNE_FREQUENCY)
      vTaskDelay((SENDYNE_FREQUENCY - (millis() - lastTime)) / portTICK_PERIOD_MS);
  }
}
```

```
//Gestione slave
void loop()
{
    lastTime = millis();
    if (command == BMS_BALANCE_CMD)
    {
      balancingMode();
      #ifdef DEBUG
      Serial.println("Balancing");
      #endif
    }
    else if (command == BMS_SLEEP_CMD)
    {
      sleepMode();
    }
    else if (command == BMS_DEBUG_CMD)
    {
      debugMode();
    }
    else if (command == BMS_NORMAL_PRINT_CMD)
    {
      normalPrintMode();
    }
    else
    {
      normalMode();
    }
    if (Serial.available() > 0)
      readSerial();
}
```

Listing A.1 bmshost2021.ino

```
#ifndef BMS_CONFIG_H
#define BMS_CONFIG_H
#include <Arduino.h>
#include <FreeRTOS.h>
#include <task.h>
#include <due_can.h>

//#define DEBUG         //enable debug print
// communication frequencies
#define SERIAL_COMMUNICATION_FREQUENCY 115200
#define INTERNAL_CAN_COMMUNICATION_FREQUENCY CAN_BPS_250K
#define SENDYNE_CAN_COMMUNICATION_FREQUENCY CAN_BPS_500K
#define EXTERNAL_CAN_COMMUNICATION_FREQUENCY CAN_1000KBPS
#define SENDYNE_FREQUENCY 10 //milliseconds
#define PWM_PERIOD_PIN_35 100000 // 1000usec fan definition
// pin definitions
#define SDC_IN 9
#define MCP_ENABLE_PIN 23
#define MCP_RS_PIN 22
#define BMS_FAULT 38
#define CHARGER_PIN 36
#define FAN_PIN 35
```

```
// command definitions
#define BMS_NORMAL_CMD "N" //THE CAR IS RACING,  THE BATTERY IS DISCHARGING
#define BMS_BALANCE_CMD "B"  //THE BATTERY IS CHARGING
#define BMS_SLEEP_CMD "S" //THE CAR IS OFF
#define BMS_DEBUG_CMD "D"
#define BMS NORMAL PRINT CMD "P"
```

```
// slave operation definitions
#define normal_mode 0x000
#define balancing_mode 0x080
#define sleep_mode 0x040
#define debug_mode 0x0C0
// external (car side) CAN definitions
#define DATA_VOLTAGE_ID 0x53
#define DATA_TEMP_ID 0x54
#define DATA_SENDYNE 0X50
#define TRANSCEIVER_STANDBY_PIN 22 //LOW TO ACTIVATE THE TRANSCEIVER(Sn65hvd234)
#define SLAVE_SELECT_PIN 12
// sendyne CAN definitions
#define REQUEST_ID 0xA100201
#define RESPONSE_ID 0xA100200
#define CURRENT_REG 0x20
#define VOLTAGE_REG 0X60
#define FAULT_FRAME_ID 0x020
#define SENDYNE_MSG_ID 0x050
// battery architecture constants
#define NUMBER_OF_SEGMENTS 8
#define NUMBER_OF_SLAVES  (NUMBER_OF_SEGMENTS*3)
#define NUMBER_OF_VS_IN_SLAVE  6  // voltages per slave
#define NUMBER_OF_TS_IN_SLAVE  2  // temperatures per slave
```

```
struct Slave
{
    byte id;
    uint16_t voltages[NUMBER_OF_VS_IN_SLAVE];
    uint16_t temperatures[NUMBER_OF_TS_IN_SLAVE];
};
// battery operation limits
#define MAX_CELL_VOLTAGE 4200  // V
#define MIN_CELL_VOLTAGE 3100  // V
#define MAX_CHARGE_CELL_TEMP  45  // C
#define MIN_CHARGE_CELL_TEMP  0  // C, used to detect erroneous readings
#define MAX_DISCHARGE_CELL_TEMP  60  // C
#define MIN_DISCHARGE_CELL_TEMP  -20  // C, used to detect erroneous readings
// communication protocol constants
#define RX_VOLTAGE_TIMEOUT  500 //milliseconds
#define RX_TEMP_TIMEOUT  500 //milliseconds
// balancing constants
#define TIME_TO_BALANCE  15000//15 seconds
#endif //BMS_DATA_H
```

Listing A.2 config.h

```
#ifndef BMS_DATA_H
#define BMS_DATA_H
#include "config.h"
#include <mcp2515.h>
#include <pwm_defs.h>
#include <pwm_lib.h>
using namespace arduino_due::pwm_lib;
// command definitions
String command = "";
String tmpCommand = "";
bool received_command = true;
// internal (battery side) CAN definitions
CAN_FRAME *first_frame;
CAN_FRAME *second_frame;
int first_frame_received = -1;
int second_frame_received = -1;
// sendyne CAN definitions
CAN_FRAME req_current, req_voltage;
byte number_recived_frame=0;
bool receaving_messagies=false;
byte sendyne_counter=0;
float sendyne_tot_current=0;
// external (car side) CAN definitions
MCP2515* mcp2515;
struct can_frame can_msg_sendyne;
//fan pwm
pwm<pwm_pin::PWMH0_PC3> pwm_pin35;
```

```
//SHUTDOWN FOR FAULT
bool fault;
unsigned int fault_detected_time;
// slaves
struct Slave slaves[NUMBER_OF_SLAVES];
//Cell proprieties
uint16_t max_cell_voltage;
uint16_t min_cell_voltage;
uint16_t max_charge_cell_temp;
uint16_t max_discharge_cell_temp;
uint16_t min_charge_cell_temp;
uint16_t min_discharge_cell_temp;
uint16_t max_voltage;//max voltage in slaves
uint16_t min_voltage;//min voltage in slaves
uint16_t mean_voltage;//mean voltage in slaves
uint16_t max_temp;//max temp in slaves
uint16_t min_temp;//min temp in slaves
uint16_t mean_temp;//mean temp in slaves
// current operation mode
byte current_operation_mode;
byte slave_sleep_ack[NUMBER_OF_SLAVES]={};
byte number_of_ack=0;
uint64_t volt_mask[NUMBER_OF_VS_IN_SLAVE]={0x3FF,0xFFC00,0x3FF00000,0xFFC0000000,0x3FF0000000000,0xFFC000000000000};
#endif //BMS_DATA_H
```

Listing A.3 data.h

```
#include "setup.h"
void scaleCellVoltage() {
  //Conversion from bq format to voltage
  max_cell_voltage=(uint16_t)(MAX_CELL_VOLTAGE-3000)/1.2;
  min_cell_voltage=(uint16_t)(MIN_CELL_VOLTAGE-3000)/1.2;
}
void scaleCellTemp() {
  //Conversion from bq format to °C
  max_charge_cell_temp=(uint16_t)(((MAX_CHARGE_CELL_TEMP+27)/0.2));
  max_discharge_cell_temp=(uint16_t)(((MAX_DISCHARGE_CELL_TEMP+27)/0.2));
  min_charge_cell_temp=(uint16_t)(((MIN_CHARGE_CELL_TEMP+27)/0.2));
  min_discharge_cell_temp=(uint16_t)(((MIN_DISCHARGE_CELL_TEMP+27)/0.2));
}
void setupPins() {
  // set pin modes
  pinMode(SLAVE_SELECT_PIN, OUTPUT);
  pinMode(MCP_ENABLE_PIN, OUTPUT);
  pinMode(MCP_RS_PIN, OUTPUT);
  pinMode(BMS_FAULT, OUTPUT);
  pinMode(CHARGER_PIN, INPUT);
  // write initial pin states
  digitalWrite(SLAVE_SELECT_PIN, HIGH);
  digitalWrite(BMS_FAULT, LOW);
  digitalWrite(MCP_ENABLE_PIN, HIGH);  // enable MCP2515
  digitalWrite(MCP_RS_PIN, LOW);
}
```

```
void setupInternalCAN() {
  Can1.begin(INTERNAL_CAN_COMMUNICATION_FREQUENCY);
  Can1.setBigEndian(true);   // operate in big endian
  Can1.setNumTXBoxes(1);   // use 1 mailbox (out of 8 on the Arduino Due) for transmission
  // set up the remaining 4 CAN mailboxes to receive messages from the slaves
  int fault_MB = Can1.setRXFilter(FAULT_FRAME_ID, 0x7E0, false);  // data and status messages from BMSs 1, 7, 13 and 19
  int standard_MB = Can1.setRXFilter(0x700, 0x700, false);  // data and status messages from BMSs 2, 8, 14 and 20
  int ack_MB = Can1.setRXFilter(0x500, 0x700, false);
  Can1.setCallback(fault_MB, gotFaultFrame);
  Can1.setCallback(standard_MB, gotFrameFromSlave);
  Can1.setCallback(ack_MB, gotAckFrame);
}
void setupSlaves() {
  for (int i = 0; i < NUMBER_OF_SLAVES; i++)
    slaves[i].id = i;  // assign right ID
}
void setupExternalCAN() {
  SPI.begin();
  pinMode(TRANSCEIVER_STANDBY_PIN,OUTPUT);
  digitalWrite(TRANSCEIVER_STANDBY_PIN,LOW);
  mcp2515=new MCP2515(SLAVE_SELECT_PIN);
  mcp2515->reset();
  mcp2515->setBitrate(EXTERNAL_CAN_COMMUNICATION_FREQUENCY);
  mcp2515->setNormalMode();
}
```

```
void setupSendyneCAN(){
  Can0.begin(SENDYNE_CAN_COMMUNICATION_FREQUENCY);
  Can0.setNumTXBoxes(1);   // use 1 mailbox (out of 8 on the Arduino Due) for transmission
  int data_MB = Can0.setRXFilter(RESPONSE_ID, RESPONSE_ID, true);
  Can0.setCallback(data_MB, gotSendyneFrame);
  req_current.id = REQUEST_ID;
  req_current.extended=true;
  req_current.length=1;
  req_current.data.bytes[0]=CURRENT_REG;
  req_voltage.id = REQUEST_ID;
  req_voltage.extended=true;
  req_voltage.length=1;
  req_voltage.data.bytes[0]=VOLTAGE_REG;
  can_msg_sendyne.can_id = SENDYNE_MSG_ID;
  can_msg_sendyne.can_dlc = 8;
}
void setupSerial()
{
  while (!Serial);
  Serial.begin(SERIAL_COMMUNICATION_FREQUENCY);
}
```

Listing A.4 setup.cpp

```
#include "can_managment.h"
void gotFaultFrame(CAN_FRAME *frame)
{
  sdcOpen();
 // extract data from frame
  byte Id_slave = (byte)(frame->id & 0x01F);
  byte isOver = frame->data.bytes[0] & 0x80;
  byte isTemp= frame->data.bytes[0]& 0x40;
  byte Id_cella = frame->data.bytes[0] & 0x3F ;
  faultSerialMessage(Id_slave,isOver,isTemp,Id_cella);
  faultLoggerMessage(Id_slave,isOver,isTemp,Id_cella);
}
```

```
//funzione ad interrupt legata alla can che riceve i pacchetti dati dagli slave e li salva nel vettore di struct salves
void gotFrameFromSlave(CAN_FRAME *frame)
{
    byte id = (byte)(frame->id & 0x01F);  // get BMS number from received CAN packet ID
    if((frame->id & 0x080) == 0x080)//temperatures
    {
       slaves[id].temperatures[0] =(uint16_t)(frame->data.value & 0x3FF);
       frame->data.value =frame->data.value >>10;
       slaves[id].temperatures[1] = (uint16_t)(frame->data.value & 0x3FF);
       second_frame_received = id;  // to confirm reception

    }else//voltages
    {
      for(int i=0;i<NUMBER_OF_VS_IN_SLAVE;i++)
      {
        slaves[id].voltages[i] = (uint16_t)(frame->data.value & 0x3FF);
        frame->data.value =frame->data.value >>10;
      }
      first_frame_received = id;  // to confirm reception

    }
}
void gotAckFrame(CAN_FRAME *frame)
{
   byte Id_slave = (byte)(frame->id & 0x01F);
   slave_sleep_ack[Id_slave]=1;
   number_of_ack++;
}
```

```
void gotDebugFrame(CAN_FRAME *frame)
{
#ifdef DEBUG
   Serial.println("RECEIVING DEBUG FRAME FROM SLAVE");
   byte Id_slave = (byte)(frame->id & 0x01F);
   Serial.print(Id_slave+ "  ");
   Serial.println(frame->data.low,HEX);
#endif
}
```

Listing A.5 can_management.cpp

```
#include "can_managment_logger.h"
void faultLoggerMessage(uint8_t Id_slave,uint8_t isOver,uint8_t isTemp,uint8_t Id_cella)
{
    uint8_t temp=0;
    temp = isOver<<7;
    temp |= isTemp<<6;
    temp |= Id_cella;
    struct can_frame outgoing;
    outgoing.can_id = 0x1F & Id_slave;
    outgoing.can_dlc = 1;
    outgoing.data[0] = temp;
    mcp2515->sendMessage(&outgoing);
}
void sendDataToCentralina(uint16_t max_volt,uint16_t mean_volt,uint16_t min_volt,uint16_t max_temp,uint16_t mean_temp,uint16_t min_temp)
{
   #ifdef DEBUG
    Serial.print("MAX VOlT:");
    Serial.println(max_volt);
    Serial.print("MIN VOLT: ");
    Serial.println(min_volt);
    Serial.print("MEAN VOLT: ");
    Serial.println(mean_volt);
    #endif
    unsigned char msg[8];
    unsigned char msg1[8];
    struct data
    {
      uint16_t one;
      uint16_t two;
      uint16_t three;
    };
    typedef struct data Data;
    Data* Msg=(Data*) msg;
    //SEND VOLTAGE DATA
    struct can_frame outgoing;
    outgoing.can_id = 0x53;
    outgoing.can_dlc = 6;
    Msg->one=max_volt;
    Msg->two=min_volt;
    Msg->three=mean_volt;
    for(int i = 0 ; i < 8; ++i)
      outgoing.data[i] =msg[i];
    mcp2515->sendMessage(&outgoing);
    //SEND TEMP DATA
    struct can_frame outgoing1;
    outgoing1.can_id = 0x54;
    outgoing1.can_dlc = 6;
    Data* Msg1=(Data*) msg1;
    Msg1->one=max_temp;
    Msg1->two=min_temp;
    Msg1->three=mean_temp;
    for(int i = 0 ; i < 8; ++i)
      outgoing1.data[i] =msg1[i];
    mcp2515->sendMessage(&outgoing1);
}
```

Listing A.6 can_managment_logger.cpp

```
#include "can_managment_sendyne.h"
void gotSendyneFrame(CAN_FRAME *frame){
  if(frame->data.bytes[0]==CURRENT_REG){
    can_msg_sendyne.data[0] = (frame->data.bytes[1]);
    can_msg_sendyne.data[1] = (frame->data.bytes[2]);
    can_msg_sendyne.data[2] = (frame->data.bytes[3]);
    can_msg_sendyne.data[3] = ((frame->data.bytes[4]) << 0);
    #ifdef DEBUG
    uint32_t reassembled_data= 0;
    reassembled_data |= ((uint32_t)(frame->data.bytes[1]) << 24);
    reassembled_data |= ((uint32_t)(frame->data.bytes[2]) << 16);
    reassembled_data |= ((uint32_t)(frame->data.bytes[3]) << 8);
    reassembled_data |= ((uint32_t)(frame->data.bytes[4]) << 0);
    float sendyne_current = (-1)*(int32_t)(reassembled_data) / 1000.0;
    Serial.print("sendyne current [mA]: ");
    Serial.println(sendyne_current);
   #endif
  }
```

```
  else if(frame->data.bytes[0]==VOLTAGE_REG){
    can_msg_sendyne.data[4] = (frame->data.bytes[1]);
    can_msg_sendyne.data[5] = (frame->data.bytes[2]);
    can_msg_sendyne.data[6] = (frame->data.bytes[3]);
    can_msg_sendyne.data[7] = (frame->data.bytes[4]);
    #ifdef DEBUG
    uint32_t reassembled_data= 0;
    reassembled_data |= ((uint32_t)(frame->data.bytes[1]) << 24);
    reassembled_data |= ((uint32_t)(frame->data.bytes[2]) << 16);
    reassembled_data |= ((uint32_t)(frame->data.bytes[3]) << 8);
    reassembled_data |= ((uint32_t)(frame->data.bytes[4]) << 0);
    float sendyne_voltage = (-1)*(int32_t)(reassembled_data) / 1000.0;
    Serial.print("sendyne voltage [mV]: ");
    Serial.println(sendyne_voltage);
   #endif
    number_recived_frame++;
    Can0.sendFrame(req_current);
  }
  if(number_recived_frame==2){
    mcp2515->sendMessage(&can_msg_sendyne);
    number_recived_frame=0;
    receaving_messagies=true;
  }
}
void requestSendyneData()
{
  Can0.sendFrame(req_voltage);
}
```

Listing A.7 can_management_sendyne.cpp

```
#include "manager.h"
///////////
//utils
///////////
boolean areDataInRange()
{
  bool ret=true;
  uint8_t i=0;
  uint8_t v=0;
  uint8_t t=0;
  String toSend;
  uint32_t total_voltage=0;
  max_voltage=0;//max voltage in slaves
  min_voltage=100000;//min voltage in slaves
  mean_voltage=0;//mean voltage in slaves
  uint16_t total_temp=0;
  max_temp=0;//max temp in slaves
  min_temp=100000;//min temp in slaves
  mean_temp=0;//mean temp in slaves
  while(i<NUMBER_OF_SLAVES)
  {
      while(v<NUMBER_OF_VS_IN_SLAVE)
      {
        if(slaves[i].voltages[v]<min_cell_voltage)
        {
            faultSerialMessage(i,1,1,v);
#ifdef DEBUG
            Serial.println(slaves[i].voltages[v]+ " "+ min_cell_voltage);
#endif
            faultLoggerMessage(i,1,1,v);
            ret= false;

        }
        if (slaves[i].voltages[v]>max_cell_voltage)
        {
            faultSerialMessage(i,0,1,v);
#ifdef DEBUG
            Serial.println(slaves[i].voltages[v]+ " "+ max_cell_voltage);
#endif
            faultLoggerMessage(i,0,1,v);
            ret= false;
        }

        if(max_voltage < slaves[i].voltages[v])
            max_voltage=slaves[i].voltages[v];
        if(min_voltage > slaves[i].voltages[v])
            min_voltage=slaves[i].voltages[v];
        total_voltage+=slaves[i].voltages[v];
        v++;
      }
#ifdef DEBUG
      Serial.print("tot voltage: ");
      Serial.println(total_voltage);
#endif
      v=0;
```

```
      while(t<NUMBER_OF_TS_IN_SLAVE)
      {
        switch(current_operation_mode)
        {
          case(balancing_mode):
            if(slaves[i].temperatures[t]<min_charge_cell_temp)
            {

              faultSerialMessage(i,1,0,t);
              faultLoggerMessage(i,1,0,t);
              ret=false;
            }
            if(slaves[i].temperatures[t]>max_charge_cell_temp)
            {
              faultSerialMessage(i,0,0,t);
              faultLoggerMessage(i,0,0,t);
              ret= false;
            }
            break;
          case(normal_mode):
            if(slaves[i].temperatures[t]<min_discharge_cell_temp)
            {
                faultSerialMessage(i,1,0,t);
                faultLoggerMessage(i,1,0,t);
                ret=false;
            }
            if( slaves[i].temperatures[t]>max_discharge_cell_temp)
            {
                 faultSerialMessage(i,0,0,t);
                 faultLoggerMessage(i,0,0,t);
                 ret=false;
            }
            break;
          default:
            break;
        }
        if(max_temp < slaves[i].temperatures[t])
            max_temp=slaves[i].temperatures[t];

        if(min_temp > slaves[i].temperatures[t])
            min_temp=slaves[i].temperatures[t];
        total_temp+=slaves[i].temperatures[t];
        t++;
      }
      t=0;
      i++;
  }
  mean_voltage=(uint16_t)(total_voltage/(NUMBER_OF_SLAVES*NUMBER_OF_VS_IN_SLAVE));
  mean_temp=(uint16_t)(total_temp/(NUMBER_OF_SLAVES*NUMBER_OF_TS_IN_SLAVE));
  return ret;
}
//////////////////
//class definition
//////////////////
void sendOperationModeRequest(byte operation_mode)
{
    CAN_FRAME outgoing;
    outgoing.id = 0x41F | operation_mode;
    outgoing.extended = 0;
    outgoing.length = 0;
#ifdef DEBUG
    Serial.print("Sending this opertation mode==>");
    Serial.println(outgoing.id);
#endif
    Can1.sendFrame(outgoing);  // send operation mode request to i-th BMS
    current_operation_mode=operation_mode;
}
bool isChargeCompleted(Slave slave)
{

  for (int i = 0; i < NUMBER_OF_VS_IN_SLAVE; i++) {
    if (slave.voltages[i] >= MAX_CELL_VOLTAGE - 1) {
      return true;
    }
  }
  return false;
}
bool isChargeCompleted()
{
   for (int i = 0; i < NUMBER_OF_SLAVES; i++) {
    if (isChargeCompleted(slaves[i])) {
      return true;
    }
  }
  return false;
}
```

```
void normalPrintMode()
{
  if(received_command==true)
  {
    received_command=false;
    sendOperationModeRequest(normal_mode);
 }
  monitorBattery();
  printData();
}
void normalMode()
{
  if(received_command==true)
  {
```

```
    received_command=false;
    sendOperationModeRequest(normal_mode);
  }
  monitorBattery();
}
void balancingMode()
{
    CAN_FRAME outgoing;
    outgoing.id = 0x41F | balancing_mode;
    outgoing.extended = 0;
    outgoing.length = 2;
    outgoing.data.s0=(uint16_t)(convertCellVoltage((int)mean_voltage)* ( 16383.0 / 6250.0));
    Can1.sendFrame(outgoing);  // send operation mode request to i-th BMS
#ifdef DEBUG
    Serial.print ("media:  ");
    Serial.println ((uint16_t)convertCellVoltage((int)mean_voltage));
#endif
    current_operation_mode=balancing_mode;
    //Wait for balance
    unsigned long lastTime=millis();
    while(millis()-lastTime<TIME_TO_BALANCE && command==BMS_BALANCE_CMD)
    {
        Can1.sendFrame(outgoing);  // send operation mode request to i-th BMS
        delay(3000);
        if (Serial.available() > 0)
          readSerial();
    }
    sendOperationModeRequest(normal_mode);
    lastTime=millis();
    while(millis()-lastTime<20000 && command==BMS_BALANCE_CMD)
```

```
    {
      monitorBattery();
      printData();
      if (Serial.available() > 0)
          readSerial();
    }
    if (isChargeCompleted())// check if charge is completed
    {
      sdcOpen();
      command=BMS_NORMAL_CMD;
    }
}
void sleepMode()
{
  if(received_command==true)
  {
    received_command=false;
    sendOperationModeRequest(sleep_mode);
  }
    long lastTime = millis();
    //aspetto ricezione degli ack dagli slave
    while (millis() - lastTime < 1000 && number_of_ack<NUMBER_OF_SLAVES);
    //se non ricevo tutti gli ack dagli slave controllo chi non me lo ha mandato
    if (number_of_ack<NUMBER_OF_SLAVES)
    {
      for (int i = 0; i < NUMBER_OF_SLAVES; i++) {
        if ( slave_sleep_ack[i]!=1) {
          sendSerial("Slave number "+String(i)+ " not in sleep mode");
        }
      }
```

```
      sdcOpen();
    }
    else
      sendSerial("All slave in sleep mode");

}
void debugMode()
{
  sendOperationModeRequest(debug_mode);
  long lastTime = millis();
}
```

```
223 void monitorBattery()
224 {
225   //Send Start_conversion_voltage to all the slaves
226    CAN_FRAME outgoing;
227    outgoing.id = 0x61F;
228    outgoing.extended = 0;
229    outgoing.length = 0;
230    Can1.sendFrame(outgoing);  // send operation mode request to i-th BMS
231   //Request data from all the slaves
232    long lastTime = millis();
233   for (byte id = 0; id < NUMBER_OF_SLAVES; id++)
234   {
235     //request voltage
236     requestVoltageFromBMS(id);
237     while ((millis()-lastTime<RX_VOLTAGE_TIMEOUT) && first_frame_received != id);
238     if (first_frame_received != id) {
239       sdcOpen();
240       sendSerial("slave "+ String(id)+" doesn't send voltage");
241       faultLoggerMessage(id,1,1,0xFF);
242     }  // i-th BMS not responding after timeout
243     //request temperatures
244     first_frame_received=-1;
245     requestTempFromBMS(id);
246     while ((millis()-lastTime<RX_TEMP_TIMEOUT) && second_frame_received != id);
247     if (second_frame_received != id) {
248       sdcOpen();
249       sendSerial("slave "+ String(id)+" doesn't send temp");
250       faultLoggerMessage(id,1,1,0xFF);
251     }  // i-th BMS not responding after timeout
252     second_frame_received=-1;
253   }
254   if(!areDataInRange())
255     sdcOpen();
256   sendDataToCentralina(max_voltage,mean_voltage,min_voltage,max_temp,mean_temp,min_temp);
257   setFan(max_temp);
258 }
259
260 void requestVoltageFromBMS(byte slave_id)
261 {
262   CAN_FRAME outgoing;
263   outgoing.id = 0x700 | slave_id;
264   outgoing.extended = 0;
265   outgoing.length = 0;
266   Can1.sendFrame(outgoing);  // send request
267 }
268 void requestTempFromBMS(byte slave_id)
269 {
270   CAN_FRAME outgoing;
271   outgoing.id = 0x780 | slave_id;
272   outgoing.id = 0x780 | slave_id;
273   outgoing.extended = 0;
274   outgoing.length = 0;
275   Can1.sendFrame(outgoing);  // send request
276 }
277
278 void setFan(uint16_t temp)
279 {
280   //semplice parabola passante per tre punti dove 30° sono 10%, 40° sono 30% e 55 sono 90%
281   float converted_temp = (float) ((0.2*temp)-27);
282   float pwm =10.0;
283   if(converted_temp<30)
284     pwm=10;
285   else if (converted_temp>55)
286     pwm=90;
287   else
288     pwm=((0.08)*(converted_temp*converted_temp)) -(3.6*(converted_temp)) +46;
289   #ifdef DEBUG
290   Serial.println(converted_temp);
291   Serial.println(pwm);
292   Serial.println((uint32_t)(PWM_PERIOD_PIN_35*(pwm/100)));
293   #endif
294   pwm_pin35.set_duty((uint32_t)(PWM_PERIOD_PIN_35*(pwm/100)));
295 }
```

Listing A.8 manager.cpp

```
1 #include "sdc_manager.h"
2 void sdcOpen(){
3     #ifdef DEBUG
4     Serial.println("SDC open");
5     #endif
6     digitalWrite(BMS_FAULT, HIGH);
7 }
```

Listing A.9 sdc_manager.cpp

```
#include "serialManager.h"
void readSerial() {  // called whenever data is available in the serial buffer
  tmpCommand = char(Serial.read());
  if(tmpCommand==BMS_NORMAL_PRINT_CMD || tmpCommand==BMS_DEBUG_CMD || tmpCommand==BMS_SLEEP_CMD || tmpCommand == BMS_BALANCE_CMD || tmpCommand == BMS_NORMAL_CMD)
  {
      command=tmpCommand;
      received_command = true;
      #ifdef DEBBUG
      Serial.print("ARRIVED ");
      Serial.println(command);
      #endif
  }
}
void sendSerial(String toSend)
{
    Serial.println(toSend);
}
float convertCellVoltage(int cell_voltage_int) {
  return 3000+(1.2*cell_voltage_int);
}
float convertCellTemp(int cell_temp_int) {
  return -27+(0.2*cell_temp_int);
}
void printData()
{
  String json="";
  json="[";
  for(int i =0;i<NUMBER_OF_SLAVES;i++)
  {
    json+="{\"volt\":[";
    for(int j=0;j<NUMBER_OF_VS_IN_SLAVE;j++)
    {
        json+=String(slaves[i].voltages[j]);
        json+=",";
    }
    json.remove(json.length()-1);
    json+="],\"temp\":[";
    for(int j=0;j<NUMBER_OF_TS_IN_SLAVE;j++)
    {
      json+=String(slaves[i].temperatures[j]);
      json+=",";
    }
    json.remove(json.length()-1);
    json+="]},";
  }
  json.remove(json.length()-1);
  json+="]";
  sendSerial(json);
}
void faultSerialMessage(byte Id_slave,byte isOver,byte isTemp,byte Id_cella)
{
  String toSend="FAULT "+String(Id_slave)+" ";
  if(isOver==0)
      toSend+="Over ";
  else
      toSend+="Under ";
  if(isTemp==0)
      toSend+="Temperature ";
  else
      toSend+="Voltage ";
  toSend+=String(Id_cella);
  sendSerial(toSend);
}
```

Lisiting A.10 serialManager.cpp

```h
#ifndef BMS_CANMANAGMENT_LOGGER_H
#define BMS_CANMANAGMENT_LOGGER_H
#include "config.h"
#include <mcp2515.h>
#include <due_can.h>

void faultLoggerMessage(uint8_t Id_slave,uint8_t isOver,uint8_t isTemp,uint8_t Id_cella);
/*
 * send data to centralina
 * max,min,mean temp and voltage
 */
void sendDataToCentralina(uint16_t max_volt,uint16_t mean_volt,uint16_t min_volt,uint16_t max_temp,uint16_t mean_temp,uint16_t min_temp);

extern MCP2515* mcp2515;
extern CAN_FRAME req_current, req_voltage;
#endif
```

Listing A.11 can_management_logger.h

```h
#ifndef BMS_CANMANAGMENT_SENDYNE_H
#define BMS_CANMANAGMENT_SENDYNE_H
#include "config.h"
#include "sdc_manager.h"
#include "can_managment_logger.h"

void gotSendyneFrame(CAN_FRAME *frame);
void requestSendyneData();

extern byte number_recived_frame;
extern bool receaving_messagies;
extern struct can_frame can_msg_sendyne;
extern CAN_FRAME req_current, req_voltage;
#endif
```

Listing A.12 can_management_sendyne.h

```h
#ifndef BMS_MANAGER_H
#define BMS_MANAGER_H
#include "config.h"
#include "serialManager.h"
#include "can_managment_logger.h"
#include "sdc_manager.h"
#include <pwm_defs.h>
#include <pwm_lib.h>

using namespace arduino_due::pwm_lib;
/*
 * @param operation_mode is the mode that the battery will work
 * 0b000->nomal mode
 * 0b010->sleep
 * 0b100->balancing
 * 0b110->debug/print
 */
void sendOperationModeRequest(byte operation_mode);
bool isChargeCompleted(Slave slave);
bool isChargeCompleted();
void normalMode();
void balancingMode() ;
void balancingPrintMode() ;
void sleepMode();
void debugMode();
void normalPrintMode();
void monitorBattery();
void requestVoltageFromBMS(byte slave_id);
void requestTempFromBMS(byte BMS_number);
void sendDataToCentralina();
float PIregulator(float setpoint, float actual_value, float Kp, float Ki, float *integral, float upper_limit_aw, float lower_limit_aw, float
void setFan(uint16_t temp);
extern bool received_command;
extern byte current_operation_mode;
extern struct Slave slaves[NUMBER_OF_SLAVES];
extern int first_frame_received;
extern int second_frame_received ;
extern byte slave_sleep_ack[NUMBER_OF_SLAVES];
extern byte  number_of_ack;
extern uint16_t max_cell_voltage;
```

```
extern uint16_t min_cell_voltage;
extern uint16_t max_charge_cell_temp;
extern uint16_t max_discharge_cell_temp;
extern uint16_t min_charge_cell_temp;
extern uint16_t min_discharge_cell_temp;
extern uint16_t max_voltage;//max voltage in slaves
extern uint16_t min_voltage;//min voltage in slaves
extern uint16_t mean_voltage;//mean voltage in slaves
extern uint16_t max_temp;//max temp in slaves
extern uint16_t min_temp;//min temp in slaves
extern uint16_t mean_temp;//mean temp in slaves
extern pwm<pwm_pin::PWMH0_PC3> pwm_pin35;
#endif
```

Listing A.13 manager.h

```
#ifndef SDC_MANAGER_H
#define SDC_MANAGER_H
#include "config.h"
//sdc open
void sdcOpen();
#endif
```

Listing A.14 sdc_manager.h

```
#ifndef BMS_SERIAL_MANAGER_H
#define BMS_SERIAL_MANAGER_H
#include "config.h"
#include <Arduino.h>
void readSerial();
void sendSerial(String toSend);
void printData();
void faultSerialMessage(byte Id_slave,byte isOver,byte isTemp,byte Id_cella);
float convertCellVoltage(int cell_voltage_int);

extern String command;
extern bool received_command;
extern struct Slave slaves[NUMBER_OF_SLAVES];
extern String tmpCommand;
extern int32_t send_volt;
extern int32_t send_curr;
extern uint16_t mean_voltage;
#endif
```

Listing A.15 serialManger.h

```
#ifndef BMS_SETUP_H
#define BMS_SETUP_H
#include "can_managment.h"
#include "can_managment_sendyne.h"
#include <SPI.h>

void setupPins();
void setupSerial();
void setupInternalCAN();
void setupSlaves();
void setupExternalCAN();
void setupSendyneCAN();
void scaleCellVoltage();
void scaleCellTemp();

extern uint16_t max_cell_voltage;
extern uint16_t min_cell_voltage;
extern uint16_t max_charge_cell_temp;
extern uint16_t max_discharge_cell_temp;
extern uint16_t min_charge_cell_temp;
extern uint16_t min_discharge_cell_temp;
extern MCP2515* mcp2515;
extern CAN_FRAME req_current, req_voltage;
extern struct can_frame can_msg_sendyne
#endif //BMS_SETUP_H
```

Listing A.16 setup.h

```
#ifndef BMS_CANMANAGMENT_H
#define BMS_CANMANAGMENT_H
#include "config.h"
#include "serialManager.h"
#include "can_managment_logger.h"
#include "sdc_manager.h"

// internal (battery side) CAN definitions
void gotFaultFrame(CAN_FRAME *frame);
void gotFrameFromSlave(CAN_FRAME *frame);
void gotAckFrame(CAN_FRAME *frame);
void gotDebugFrame(CAN_FRAME *frame);

extern struct Slave slaves[NUMBER_OF_SLAVES];
extern int first_frame_received ;
extern int second_frame_received;
extern byte slave_sleep_ack[NUMBER_OF_SLAVES];
extern byte number_of_ack;
extern uint64_t volt_mask[NUMBER_OF_VS_IN_SLAVE];

#endif
```

Listing A.17 can_management.h

Appendice B

Codice Slave

```
#include <SPI.h>
#include <avr_can.h>
#include "data.h"
#include "utils.h"
#include "setup.h"
#include "bq_io.h"
#include "updater.h"
#include "CAN.h"
void setup() {
  setupPins();
  BMS_NUMBER = getBMSnumber();
  setupCAN();
  SPI.begin();
  setupDevice();
  PRR |= ( 1 << PRPSC ) || ( 1 << PRTIM1 ) || ( 1 << PRTIM0 ) || ( 1 << PRLIN ) || ( 1 << PRADC ); // enable power reduction where needed
  //visual check if bq comunication works
  gpio( true ); // turn on LED
  delay( 2 * ONE_SEC );
  gpio( false ); // turn off LED
  digitalWrite( LED_PIN, LOW );
  //visual check if BMS number is correct
  for ( uint8_t i = 1; i <= ( BMS_NUMBER ); i++ ) {
    digitalWrite( LED_PIN, HIGH );
    delay( 300 );
    digitalWrite( LED_PIN, LOW );
    delay( 300 );
  }
}
void loop()
{
}
```

Listing B.1 bms_slave.ino

```
#ifndef BMS_CAN_H
#define BMS_CAN_H
#include "data.h"
#include "utils.h"
#include "bq_io.h"
#include "updater.h"
int sendErrorFrame( bool isOver, bool isVoltage, uint8_t number ) {
  CAN_FRAME outgoing;
  outgoing.id =0x020;
  outgoing.id |= BMS_NUMBER ;
  outgoing.extended = false;
  outgoing.length = 1;
  outgoing.data.bytes[0] = 0;
  outgoing.data.bytes[0] |= isOver << IS_OVER_BIT;
  outgoing.data.bytes[0] |= isVoltage << IS_VOLTAGE_BIT;
  outgoing.data.bytes[0] |= number;
  Can0.sendFrame( outgoing );
}
void sendOperationModeACK( byte operation_mode ) {
  CAN_FRAME outgoing;
  outgoing.id = MODE_ACK_ID |  (operation_mode) | BMS_NUMBER;
  outgoing.extended = false;
  outgoing.length = 0;
  Can0.sendFrame( outgoing ); // send ACK
}
void gotVoltRequest( CAN_FRAME *frame ){

  CAN_FRAME outgoing;
  outgoing.id = frame->id;
  outgoing.extended = false;
  outgoing.data.value=0;
  outgoing.length = 8;
  getVoltagesPacket(&outgoing);
```

```
  }
void gotTempRequest( CAN_FRAME *frame ){
  CAN_FRAME outgoing;
  outgoing.id = frame->id;
  outgoing.extended = false;
  outgoing.data.value=0;
  outgoing.length = 3;
  getTemperaturesPacket(&outgoing);
}
void gotDataRequest( CAN_FRAME *frame ) {
  while (digitalRead(DRDY_PIN) == 0);
  digitalWrite(CONV_PIN, LOW);
  updateAll();
  CAN_FRAME outgoing;
  if((frame->id & 0x080) == 0x080){ //temperatures are required
    outgoing.id = 0x780 | BMS_NUMBER;
    outgoing.extended = false;
    outgoing.length = 3;
    getTemperaturesPacket(&outgoing);
  } else { //voltages are required
    outgoing.id = 0x700 | BMS_NUMBER;
    outgoing.extended = false;
    outgoing.length = 8;
    getVoltagesPacket(&outgoing);
  }
  Can0.sendFrame( outgoing );
}
void gotConversionRequest( CAN_FRAME *frame ) {
  startConversion();
}
void gotStatusRequest( CAN_FRAME *frame ) {
  uint8_t mode = getRequestedMode(frame);
  switch ( mode )
```

```
  {
    case NORMAL_MODE:
      {
        wakeUp();
        break;
      }
    case BALANCING_MODE:
      {
        cellBalancing( frame->data.s0 );
        break;
      }
    case SLEEP_MODE:
      {
        putSleep();
        sendOperationModeACK( mode );
        break;
      }
    case DEBUG_MODE:
      {
        debug();
        sendOperationModeACK( mode );
        break;
      }
    default:
      {
        wakeUp();
        break;
      }
  }
}
void faultInterrupt() {
  byte faultStatus = getFaults();
  byte cuv=0;
```

```
  if((faultStatus & 0b00000001)==0b00000001){ //OverVoltage fault
    byte cov = getCOV();
    sendErrorFrame( 0 , 1, cov );
  }
  if((faultStatus & 0b00000010)==0b00000010){ //UnderVoltage fault
    cuv = getCUV();
    sendErrorFrame( 1 , 1, cuv );
  }
  bqClearFaults();
}
void drdyInterrupt() {
  digitalWrite( CONV_PIN, LOW );  //FINISH CONVERSION
  gpio(0);  //VISUAL FEEDBACK
  updateAll();
}
#endif
```

Listing B.2 CAN.h

```
#ifndef BMS_IO_H
#define BMS_IO_H
#include "data.h"
#include "utils.h"
void bqWrite( byte device_address, byte reg_address, byte reg_data ) {
  delayMicroseconds( 5 );
  digitalWrite( SLAVE_SELECT_PIN, LOW ); // Take the SS pin low to select the chip
  delayMicroseconds( 5 );
  SPI.setDataMode( SPI_MODE1 );
  SPI.beginTransaction( SPISettings( 50000, MSBFIRST, SPI_MODE1 ) );
  // Shift the device bit and set bit 0
  byte logical_address = ( device_address << 1 ) | 0x01;
  byte crc_input[3] = {logical_address, reg_address, reg_data};
  // Send and receive SPI
  SPI.transfer( logical_address );
  SPI.transfer( reg_address );
  SPI.transfer( reg_data );
  SPI.transfer( pec( crc_input ) );
  delayMicroseconds( 5 );
  digitalWrite( SLAVE_SELECT_PIN, HIGH ); // Take the SS pin high to de-select the chip
  SPI.endTransaction();
}
byte *bqRead( byte device_address, byte reg_address, byte length )    {
  delayMicroseconds( 5 );
  digitalWrite( SLAVE_SELECT_PIN, LOW ); // Take the SS pin low to select the chip
  delayMicroseconds( 5 );
  SPI.beginTransaction( SPISettings( 50000, MSBFIRST, SPI_MODE1 ) );
  // Shift the device bit and clear bit 0
  byte logical_address = device_address << 1;
  logical_address &= 0b11111110;
  // Create buffer for receivedData and clear it
  static byte received_data[20];
  memset( received_data, 0, sizeof( received_data ) );
  // send and receive SPI
  SPI.transfer( logical_address );
  SPI.transfer( reg_address );
  SPI.transfer( length );
  delayMicroseconds( 1 );
  for ( uint8_t i = 0; i < length + 1; i++ ) {
    received_data[i] = SPI.transfer( 0x00 );
  }
  delayMicroseconds( 5 );
  digitalWrite( SLAVE_SELECT_PIN, HIGH ); // Take the SS pin high to de-select the chip
  SPI.endTransaction();
  return received_data;
}
void bqClearAlerts() {
  // clear alert bit in device status register
  byte *value = bqRead( DEVICE_ADDR, DEVICE_STATUS, 1 );
  value[0] &= 0b11011111;  // clear alert bit
  bqWrite( DEVICE_ADDR, DEVICE_STATUS, value[0] );
  byte value_res = 0x00;  // Write 0's ALERT_STATUS_REG register
  bqWrite( DEVICE_ADDR, ALERT_STATUS, value_res );
}
void bqClearFaults() {
  // clear fault bit in device status register
  byte *value = bqRead( DEVICE_ADDR, DEVICE_STATUS, 1 );
  value[0] &= 0b10111111;  // clear fault bit
  bqWrite( DEVICE_ADDR, DEVICE_STATUS, value[0] );
  byte value_res = 0x00;  // Write 0's FAULT_STATUS_REG register
  bqWrite( DEVICE_ADDR, FAULT_STATUS, value_res );
}
void resetDevice() {
  bqWrite( BROADCAST_ADDR, RESET, BQ76PL536_RESET ); // write reset code to reset register
  bqClearFaults();
  bqClearAlerts();
}
void gpio( bool state ) {
  byte *value = bqRead( DEVICE_ADDR, IO_CONTROL, 2 );
  if ( state ) {
    value[0] |= 0b01000000;
  }
  else {
    value[0] &= 0b10111111;
  }
  bqWrite( DEVICE_ADDR, IO_CONTROL, value[0] );
}
#endif
```

Lisitng B.3 bq_io.h

```
#ifndef BMS_DATA_H
#define BMS_DATA_H
// communication frequency
#define CAN_COMMUNICATION_FREQUENCY CAN_BPS_250K
// pin definitions CHECK ATmega32m1 DOCUMENTATION
#define CONV_PIN 10
#define DRDY_PIN 9
#define FAULT_PIN A5
#define ALERT_PIN 8
#define FAULT_PIN_INTERRUPT PCINT14
#define ALERT_PIN_INTERRUPT PCINT3
#define DRDY_PIN_INTERRUPT PCINT4
#define SLAVE_SELECT_PIN 1
#define LED_PIN A2
// battery operation definitions
#define MODE_ID_MASK 0x0E0
#define NORMAL_MODE 0x00 //
#define SLEEP_MODE 0x40 // -> ack richiesto
#define BALANCING_MODE 0x80 // -> nel payload mettere la tensione
#define DEBUG_MODE 0xC0
// CAN IDs definitions
#define MODE_REQUEST_ID_MASK 0x71F
#define MODE_REQUEST_ID 0x41F
#define MODE_ACK_ID 0x500
#define MODE_ACK 0x0E0
#define CONVERSION_REQUEST_ID_MASK 0x700
#define CONVERSION_REQUEST_ID 0x600
#define DATA_REQUEST_ID_MASK 0x700
#define DATA_REQUEST_ID 0x700
#define CAN_ID_MASK 0x7E0
#define BROADCAST_NUMBER 0b00011111
#define BMS_NUMBER_MASK 0x01F
#define ERROR_ID 0x000
```

```
#define ALERT_ID 0x020
#define IS_OVER_BIT 7
#define IS_VOLTAGE_BIT 6
// bq addresses definitions
#define DEVICE_ADDR 0x01
#define BROADCAST_ADDR 0x3F
#define DISCOVERY_ADDR 0x00
#define BQ76PL536_RESET 0xA5
// time constants
#define ONE_MS 1
#define ONE_SEC 1000
// bq register definitions
const byte crcTable[256] = {
  0x00, 0x07, 0x0E, 0x09, 0x1C, 0x1B, 0x12, 0x15, 0x38, 0x3F, 0x36, 0x31, 0x24, 0x23, 0x2A,
  0x2D, 0x70, 0x77, 0x7E, 0x79, 0x6C, 0x6B, 0x62, 0x65, 0x48, 0x4F, 0x46, 0x41, 0x54, 0x53, 0x5A, 0x5D,
  0xE0, 0xE7, 0xEE, 0xE9, 0xFC, 0xFB, 0xF2, 0xF5, 0xD8, 0xDF, 0xD6, 0xD1, 0xC4, 0xC3, 0xCA, 0xCD, 0x90,
  0x97, 0x9E, 0x99, 0x8C, 0x8B, 0x82, 0x85, 0xA8, 0xAF, 0xA6, 0xA1, 0xB4, 0xB3, 0xBA, 0xBD, 0xC7, 0xC0,
  0xC9, 0xCE, 0xDB, 0xDC, 0xD5, 0xD2, 0xFF, 0xF8, 0xF1, 0xF6, 0xE3, 0xE4, 0xED, 0xEA, 0xB7, 0xB0, 0xB9,
  0xBE, 0xAB, 0xAC, 0xA5, 0xA2, 0x8F, 0x88, 0x81, 0x86, 0x93, 0x94, 0x9D, 0x9A, 0x27, 0x20, 0x29, 0x2E,
  0x3B, 0x3C, 0x35, 0x32, 0x1F, 0x18, 0x11, 0x16, 0x03, 0x04, 0x0D, 0x0A, 0x57, 0x50, 0x59, 0x5E, 0x4B,
  0x4C, 0x45, 0x42, 0x6F, 0x68, 0x61, 0x66, 0x73, 0x74, 0x7D, 0x7A, 0x89, 0x8E, 0x87, 0x80, 0x95, 0x92,
  0x9B, 0x9C, 0xB1, 0xB6, 0xBF, 0xB8, 0xAD, 0xAA, 0xA3, 0xA4, 0xF9, 0xFE, 0xF7, 0xF0, 0xE5, 0xE2, 0xEB,
  0xEC, 0xC1, 0xC6, 0xCF, 0xC8, 0xDD, 0xDA, 0xD3, 0xD4, 0x69, 0x6E, 0x67, 0x60, 0x75, 0x72, 0x7B, 0x7C,
  0x51, 0x56, 0x5F, 0x58, 0x4D, 0x4A, 0x43, 0x44, 0x19, 0x1E, 0x17, 0x10, 0x05, 0x02, 0x0B, 0x0C, 0x21,
  0x26, 0x2F, 0x28, 0x3D, 0x3A, 0x33, 0x34, 0x4E, 0x49, 0x40, 0x47, 0x52, 0x55, 0x5C, 0x5B, 0x76, 0x71,
  0x78, 0x7F, 0x6A, 0x6D, 0x64, 0x63, 0x3E, 0x39, 0x30, 0x37, 0x22, 0x25, 0x2C, 0x2B, 0x06, 0x01, 0x08,
  0x0F, 0x1A, 0x1D, 0x14, 0x13, 0xAE, 0xA9, 0xA0, 0xA7, 0xB2, 0xB5, 0xBC, 0xBB, 0x96, 0x91, 0x98, 0x9F,
  0x8A, 0x8D, 0x84, 0x83, 0xDE, 0xD9, 0xD0, 0xD7, 0xC2, 0xC5, 0xCC, 0xCB, 0xE6, 0xE1, 0xE8, 0xEF, 0xFA,
  0xFD, 0xF4, 0xF3
};
const byte DEVICE_STATUS = 0x00;  //bq registers
const byte GPAI = 0x01;
const byte VCELL[6] = { 0x03,
```

```
            0x05,
            0x07,
            0x09,
            0x0B,
            0x0D
            };
const byte TEMPERATURE[2] = { 0x0F,
              0x11
            };
const byte ALERT_STATUS = 0x20;
const byte FAULT_STATUS = 0x21;
const byte COV_FAULT = 0x22;
const byte CUV_FAULT = 0x23;
const byte PRESULT_A = 0x24;
const byte PRESULT_B = 0x25;
const byte ADC_CONTROL = 0x30;
const byte IO_CONTROL = 0x31;
const byte CB_CTRL = 0x32;
const byte CB_TIME = 0x33;
const byte ADC_CONVERT = 0x34;
const byte SHDW_CTRL = 0x3A;
const byte ADDRESS_CONTROL = 0x3B;
const byte RESET = 0x3C;
const byte TEST_SELECT = 0x3D;
const byte E_EN = 0x3F;
const byte FUNCTION_CONFIG = 0x40;
const byte IO_CONFIG = 0x41;
const byte CONFIG_COV = 0x42;
const byte CONFIG_COVT = 0x43;
const byte CONFIG_CUV = 0x44;
const byte CONFIG_CUVT = 0x45;
const byte CONFIG_OT = 0x46;
const byte CONFIG_OTT = 0x47;
const byte USER1 = 0x48;
const byte USER2 = 0x49;
const byte USER3 = 0x4A;
const byte USER4 = 0x4B;
const byte BATTERY_A_OV = 1;  // Error codes for alert message
const byte BATTERY_A_UV = 2;
const byte BATTERY_A_UV_WARN = 3;
const byte CELL_IMBALANCE = 4;
const byte CELL_OV = 5;
const byte CELL_UV = 6;
const byte BATTERY_A_OC = 7;  // battery
const byte BATTERY_A_OC_WARN = 8;
const byte BATTERY_B_OV = 10;
const byte BATTERY_B_UV = 11;
const byte BATTERY_B_UV_WARN = 12;
const byte BATTERY_B_OC = 13;
const byte BATTERY_B_OC_WARN = 14;
const byte TEMP_A_OT = 20;  // temperature
const byte TEMP_A_OT_WARN = 21;
const byte TEMP_B_OT = 22;
const byte TEMP_B_OT_WARN = 23;
const int MAX_CELL_VOLTAGE = 4200; // mV
const byte OV_THRESHOLD = 0x2c; // -> 4.2V -> (4.2-2.0)/0.05 = 44 = 0x2c ??????
const byte UV_THRESHOLD = 0x1c; // -> 3.5V -> (3.5-0.7)/0.1 = 28 = 0x1c
//const byte UV_THRESHOLD = 0x1f; // -> 3.8V -> (3.8-0.7)/0.1 = 31 = 0x1f
const byte OT_THRESHOLD = 0x55; // -> TS1=60C , TS2 =60C
const byte OV_TIMER = 0x05 | 0x80;  // 500ms + MSB set to 1 for milliseconds -> 500 = 0x05
const byte UV_TIMER = 0x05 | 0x80;  // 500ms + MSB set to 1 for milliseconds -> 500 = 0x05
const byte OT_TIMER = 0x64;  // 100*10ms =1s
// balancing constants
const byte CB_SAFETY_TIMER = 0x01; //|| 0x00;  // 10s + MSB set to 0 for seconds -> 10 = 0x0a
const uint8_t BALANCE_RANGE = 0;
const int BALANCING STABILIZATION TIME = 50 * ONE MS;
```

```
// battery constants
const uint8_t NUMBER_OF_BQ_DEVICES = 24;
const uint8_t NUMBER_OF_BQ_DEVICES_PER_SEGMENT = 3;
const uint8_t NUMBER_OF_CELLS_IN_BQ_DEVICE = 6;
const uint8_t NUMBER_OF_TS_IN_BQ_DEVICE = 2;
// thermistor data
const float Beta = 3435.0;
const float Rb = 1800.0;
const float Rt = 1500.0;
const float R0 = 10000.0;
// auxiliary definitions
const uint8_t ID_PINS[5] = {13, 12, 11, A7, A6};
uint8_t BMS_NUMBER;
int counter=0;
// arrays to store data
volatile struct BMS_VALUES{
  uint16_t voltages[NUMBER_OF_CELLS_IN_BQ_DEVICE];
  float temperatures[NUMBER_OF_TS_IN_BQ_DEVICE];
} bmsValues;
#endif //BMS_DATA_H
```

Lisiting B.4 data.h

```
#ifndef BMS_READ_H
#define BMS_READ_H
#include "data.h"
#include "utils.h"
uint16_t getCellVoltage( uint8_t cell_number ) {
  byte *cell_raw = bqRead( DEVICE_ADDR, VCELL[cell_number], 0x02 );
  return (cell_raw[0]<<8) + cell_raw[1];
}
float getGpaiVbatt( bool dev ) {
  byte *gpai_raw = bqRead( DEVICE_ADDR, GPAI, 0x02 );
  float val = ( ( gpai_raw[0] * 256 ) + gpai_raw[1] ) / 16383.0;
  if ( dev ) {
    return val * 33.333;    // [V]
  }
  else {
    return val * 2500;    // [mV]
  }
}
float getTS( uint8_t ts_number ) {
  byte *temp_raw = bqRead( DEVICE_ADDR, TEMPERATURE[ts_number], 2 );
  return convertRawTemperature( temp_raw );
}
byte getFaults() {
  byte *fault = bqRead( DEVICE_ADDR, FAULT_STATUS, 1 );
  return fault[0];
}
byte getStatus() {
  byte *status = bqRead( DEVICE_ADDR, DEVICE_STATUS, 1 );
  return status[0];
}
byte getCOV() {
  byte *value = bqRead( DEVICE_ADDR, COV_FAULT, 1 );
  return (value[0] & 0b00111111);
}
byte getCUV() {
  byte *value = bqRead( DEVICE_ADDR, CUV_FAULT, 1 );
  return (value[0] & 0b00111111);
}
#endif
```

Listing B.5 read.h

```
#ifndef BMS_SETUP_H
#define BMS_SETUP_H
#include "data.h"
#include "utils.h"
#include "bq_io.h"
#include "CAN.h"
#include "updater.h"
void shadowControlWrite() {  // allow writing on group3 protected registers
  bqWrite( DEVICE_ADDR, SHDW_CTRL, 0x35 );
}
void covCuvCotSetup() {    //setup secondary Cell Over/Under Voltage Protection
  shadowControlWrite();
  bqWrite( DEVICE_ADDR, CONFIG_COV, OV_THRESHOLD ); // configure cell overvoltage
  shadowControlWrite();
  bqWrite( DEVICE_ADDR, CONFIG_COVT, OV_TIMER ); // configure cell overvoltage timer
  shadowControlWrite();
  bqWrite( DEVICE_ADDR, CONFIG_CUV, UV_THRESHOLD ); // configure cell undervoltage
  shadowControlWrite();
  bqWrite( DEVICE_ADDR, CONFIG_CUVT, UV_TIMER ); // configure cell undervoltage timer
  shadowControlWrite();
  bqWrite( DEVICE_ADDR, CONFIG_OT, OT_THRESHOLD ); // configure cell overtemperature
  shadowControlWrite();
  bqWrite( DEVICE_ADDR, CONFIG_OTT, OT_TIMER ); // configure cell overtemperature timer
}
void setAddress() {
  bqWrite( DISCOVERY_ADDR, ADDRESS_CONTROL, DEVICE_ADDR ); // assign the same address to all bq devices
}
void setupBqRegister() {  // setup BQ register, power on all LEDs then after 2sec power off (visual check)
  bqWrite( DEVICE_ADDR, ADC_CONTROL, 0b00111101 ); // TS ON, GPAI ON, measure 1 to 6 cells
  shadowControlWrite();
  bqWrite( DEVICE_ADDR, FUNCTION_CONFIG, 0b01110000 ); // GPAI measure 6 cells series on VBAT to VSS
  bqWrite( DEVICE_ADDR, IO_CONTROL, 0b00000011 ); // connect TS2- and TS1- to VSS
  bqWrite( DEVICE_ADDR, CB_TIME, CB_SAFETY_TIMER ); // set Cell Balance Control Safety Timer
```

```
  covCuvCotSetup();
  bqClearAlerts();
  bqClearFaults();
}
void setupPins() {
  // set pin modes and interrupts
  pinMode( SLAVE_SELECT_PIN, OUTPUT );
  pinMode( FAULT_PIN, INPUT_PULLUP );
  pinMode( ALERT_PIN, INPUT_PULLUP );
  pinMode( DRDY_PIN, INPUT_PULLUP );
  pinMode( CONV_PIN, OUTPUT );
  pinMode( LED_PIN, OUTPUT );
  for(byte i=0;i<5;i++)
    pinMode( ID_PINS[i],INPUT_PULLUP);
  attachInterrupt(INT3, faultInterrupt, RISING );
  attachInterrupt( INT1, alertInterrupt, RISING );
  // write initial pin states
  digitalWrite( CONV_PIN, LOW );
  digitalWrite( SLAVE_SELECT_PIN, HIGH );
  digitalWrite( LED_PIN, LOW );
}
void setupCAN() {
  Can0.begin( CAN_COMMUNICATION_FREQUENCY ); // initialize CAN
  Can0.setBigEndian( true ); // operate in big endian
  Can0.setNumTXBoxes( 2 ); // use 2 mailboxes (out of 6 on the ATmega32m1) for transmission
  // set up the remaining 3 CAN mailboxes to receive messages (sintax: (id, mask, extended) )
  int mode_MB = Can0.setRXFilter( MODE_REQUEST_ID, MODE_REQUEST_ID_MASK, false ); // status requests from the host
  int conv_MB = Can0.setRXFilter( CONVERSION_REQUEST_ID, CONVERSION_REQUEST_ID_MASK, false ); // conversion requests from the host
  int data_MB = Can0.setRXFilter( DATA_REQUEST_ID | BMS_NUMBER, DATA_REQUEST_ID_MASK | BMS_NUMBER_MASK, false ); // data requests from the host
  Can0.setCallback( mode_MB, gotStatusRequest );
  Can0.setCallback( conv_MB, gotConversionRequest );
  Can0.setCallback( data_MB, gotDataRequest );
}
```

```
void setupDevice() {
  resetDevice();
  delay( ONE_MS );
  setAddress();  // 0xa5 reset 0x3c reset register 0x3f broadcast all devices
  delay( ONE_MS );
  setupBqRegister();
  delay( ONE_MS );
}
void enableAux5V() {
  byte *value = bqRead( DEVICE_ADDR, IO_CONTROL, 1 );
  value[0] |= 0b10000000;
  bqWrite( DEVICE_ADDR, DEVICE_STATUS, value[0] );
}
#endif
```

Listing B.6 setup.h

```
#ifndef BMS_UPDATER_H
#define BMS_UPDATER_H
#include "data.h"
#include "utils.h"
#include "read.h"
#include "CAN.h"
void startConversion() {
  digitalWrite( CONV_PIN, HIGH ); //HARDWARE START
  //SOFTWARE START (slower but more flexible), may add filter to select fewer channels
  //bqWrite( DEVICE_ADDR, ADC_CONVERT, 0x01 );
}
void *updateVoltages() {
  for ( uint8_t i = 0; i < NUMBER_OF_CELLS_IN_BQ_DEVICE ; i++ ) { // 6 couples of cells monitored by each bq device
    bmsValues.voltages[i] = getCellVoltage( i );
  }
}
void *updateTemperatures() {
  bmsValues.temperatures[0] = getTS( 0 );
  bmsValues.temperatures[1] = getTS( 1 );
}
void updateAll() {
  updateVoltages();
  updateTemperatures();
}
void cellBalancing( int threshold_voltage ) {
  byte cells_to_balance = 0;  // byte containing the positions of the cells to balance
  bqWrite( DEVICE_ADDR, CB_CTRL, cells_to_balance ); //reset CB_CTRL register (also Safety timer is reset by BQ)
  delay( BALANCING_STABILIZATION_TIME );
  updateVoltages();                    //read cell voltages while not balancing
  for ( uint8_t i = 0; i < NUMBER_OF_CELLS_IN_BQ_DEVICE; i++ ) {
    if ( bmsValues.voltages[i] > threshold_voltage + BALANCE_RANGE ) {
      cells_to_balance |= ( 1 << i ); // count the unbalanced cell
    }
```

```
  }
  bqWrite( DEVICE_ADDR, CB_CTRL, cells_to_balance ); // balance cells
  //the process of balancing will automatically stop after CB_SAFETY_TIMER seconds
}
void putSleep() {
  byte *value = bqRead( DEVICE_ADDR, IO_CONTROL, 1 );
  value[0] |= 0b00000100;  // set IO_CONTROL[SLEEP] bit
  value[0] &= 0b11111100;  // reset IO_CONTROL[TS1, TS2] bits
  bqWrite( DEVICE_ADDR, IO_CONTROL, value[0] );
  bqClearAlerts();  // reset ALERT[SLEEP] bit
}
void wakeUp() {
  byte *value = bqRead( DEVICE_ADDR, IO_CONTROL, 1 );
  value[0] &= 0b11111011;  // reset IO_CONTROL[SLEEP] bit
  value[0] |= 0b00000011;  // set IO_CONTROL[TS1, TS2] bits
  bqWrite( DEVICE_ADDR, IO_CONTROL, value[0] );
  bqClearAlerts();  // reset ALERT[SLEEP] bit
}
void debug() {  // turn both LEDs to identify defective slaves
  digitalWrite( LED_PIN, HIGH ); // turn on ATmega LED
  gpio( true ); // turn on bq LED
}
#endif
```

Listing B.7 updater.h

```
#ifndef BMS_UTILS_H
#define BMS_UTILS_H
#include "data.h"
uint8_t getBMSnumber() {  //Get the number of the BMS board
  uint8_t number = 0;
  for (int i = 0; i<5; i++){
    number = number<<1;
    number |= digitalRead(ID_PINS[i]);
  }
  return number;
}
uint8_t getTargetBMS(CAN_FRAME *frame) {  //get the target BMS of a CAN frame
  uint8_t number = frame->id & BMS_NUMBER_MASK;
  return number;
}
uint8_t getRequestedMode(CAN_FRAME *frame) {  //get the mode from of a mode request frame
  uint8_t mode = (frame->id & MODE_ID_MASK);
  return mode;
}
byte pec(byte crcBuffer[]) {
  byte crc = 0;
  int temp = 0;
  for (int i = 0; i < 3; i++) {
    temp = crc ^ crcBuffer[i];
    crc = crcTable[temp];
  }
  return crc;
}
uint16_t compressCellVoltage(uint16_t value) { //drop the resolution from 14 bit to 10 bit
  float voltage = ((float) value) * (6250.0 / 16383.0);
  uint16_t compressedValue = (voltage-3000) / 1.2;
  return compressedValue;
}
```

```
void getVoltagesPacket(CAN_FRAME *frame){
  frame->data.value=0;
  for ( uint8_t i = NUMBER_OF_CELLS_IN_BQ_DEVICE -1; i > 0 ; i-- ) {
    frame->data.value |= (compressCellVoltage(bmsValues.voltages[i]) & 0x03FF);
    frame->data.value = frame->data.value << 10;
  }
  frame->data.value |= (compressCellVoltage(bmsValues.voltages[0]) & 0x03FF);
}
uint16_t compressTemperature(float value) { //drop the resolution to 10 bit
  uint16_t compressedValue = (value+27) / 0.2;
  return compressedValue;
}
void getTemperaturesPacket(CAN_FRAME *frame){
  frame->data.value=0;
  for ( uint8_t i = NUMBER_OF_TS_IN_BQ_DEVICE -1; i > 0 ; i-- ) {
    frame->data.value |= (compressTemperature(bmsValues.temperatures[i]) & 0x03FF);
    frame->data.value = frame->data.value << 10;
  }
  frame->data.value |= (compressTemperature(bmsValues.temperatures[0]) & 0x03FF);
}
```

```
54 float convertRawTemperature(byte* value) {
55   //unsigned int temp_raw = (value[0] << 8) + value[1];
56   short temp_raw = (value[0] * 0x100) + value[1];
57   float ratio = (temp_raw + 2) / 33046.0;
58   float Rth = ((1 / ratio) - 1) * Rb - Rt;
59   float temperature = Beta / (log(Rth / (R0 * exp(-Beta / 298.15))));
60   return temperature - 273.15;  // from kelvin to celsius
61 }
62 #endif
```

Listing B.8 utils.h

Bibliografia

[1] RACEUP TEAM. Home page. https://www.raceup.it.

[2] GIOVANNI, M. Pagina personale DII https://www.dii.unipd.it/category/ruoli/personale-docente?key=3DA0BCA9B0515F5B41534A83DFFE827D

[3] RACE UP TEAM. sponsor https://www.raceup.it/sponsors/collaborations.html

[4] FORMULA STUDENT GERMANY, regolamento https://www.formulastudent.de/fileadmin/user_upload/all/2022/rules/FS-Rules_2022_v1.0.pdf

[5] MELASTA SLPBA843126, https://melasta.en.made-in-china.com/product/neixcbUHhNRj/China-High-Power-15c-6350mAh-3-7V-Li-Polymer-Cell-for-RC-Aircraft.html

[6] Sendyne Corp. SFP200MOD Datasheet. 250 West Broadway, New York, December 2019.

[7] Sendyne. Home page. http://www.sendyne.com/.

[8] Arduino Due. Overview. https://store.arduino.cc/products/arduino-due

[9] bq56pl536a-q1. Datasheet. https://www.ti.com/lit/ds/symlink/bq76pl536a-q1.pdf?ts=1645476289138&ref_url=https%253A%252F%252Fwww.google.com%252F

[10] ATMEGA32M1-AU. Datasheet https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8209-8-bit%20AVR%20ATmega16M1-32M1-64M1_Summary.pdf

[11] Arduino Software (IDE). Homepage. https://www.arduino.cc/en/software