Formula Electric@Berkeley BMS Technical Report

Deven Bibikar

Spring 2025

Contents

Introduction

This document aims to detail the thought-process behind the BMS diagram for my introductory project for Formula Electric. Here, you will find explanations for why I've chosen to detail my diagram as such, how certain code segments function, and other related lessons I've learned along the way. I want to thank Formula Electric @Berkeley for this opportunity. It's been fun learning about the BMS.

Chapter 1

BMS Diagram

1.1 Introduction

This section intends to detail and explain each section of the BMS State Machine Diagram. Since the diagram is quite large, it can be found at the end of the chapter.

1.2 Main States

I define the IDLE state as a state where the battery is not discharging any power. I define the DRIVE state as a state where the battery is actively discharging power.

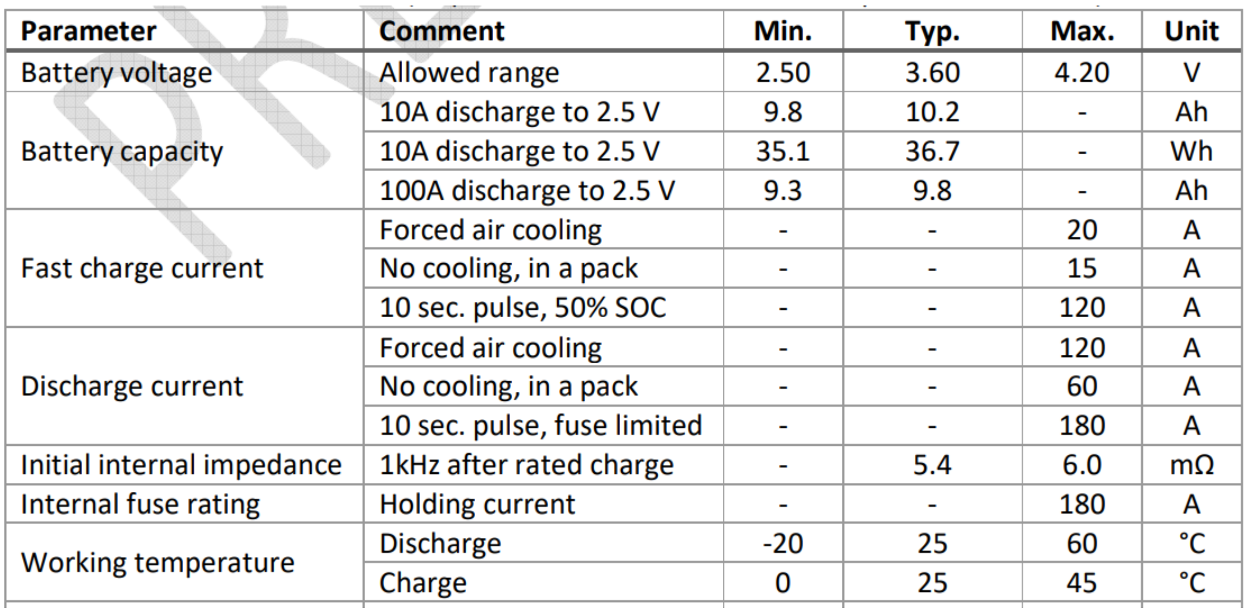
1.3 Sub-states

Checking conditions

Regardless of whether the state is in DRIVE or IDLE, temperature, voltage, current should be continuously checked to see whether they're running within operating levels. We should likewise always check whether our fuse is okay, whether our overcurrent protection is functioning alright, and whether the shutdown signal is or isn't active. If any of these properties (outside the shutdown signal) fail to meet the expected values, then we should execute a fault sequence.

If the shutdown signal is true, then the user has signaled to the machine that a shutdown should take place. In this case, the car should move from a DRIVE state into an IDLE state. The car should not enter a DRIVE state until the shutdown signal is no longer active. If at any time a shutdown process fails, fault or otherwise, the respective process should be restarted.

Sometimes items are within operating levels but do require cooling. See Figure 1.1 for when cooling should or shouldn't take place.



Parameter	Comment	Min.	Typ.	Max.	Unit
Battery voltage	Allowed range	2.50	3.60	4.20	V
Battery capacity	10A discharge to 2.5 V	9.8	10.2	-	Ah
	10A discharge to 2.5 V	35.1	36.7	-	Wh
	100A discharge to 2.5 V	9.3	9.8	-	Ah
Fast charge current	Forced air cooling	-	-	20	A
	No cooling, in a pack	-	-	15	A
	10 sec. pulse, 50% SOC	-	-	120	A
Discharge current	Forced air cooling	-	-	120	A
	No cooling, in a pack	-	-	60	A
	10 sec. pulse, fuse limited	-	-	180	A
Initial internal impedance	1kHz after rated charge	-	5.4	6.0	mΩ
Internal fuse rating	Holding current	-	-	180	A
Working temperature	Discharge	-20	25	60	°C
	Charge	0	25	45	°C

Figure 1.1: Relevant Condition Data from Battery specification sheet

Fault Shutdown Sequence

When the fault sequence is initiated, the shutdown circuit, a related item to the BMS should be run. Since this is not a state in the BMS, it is not explicitly shown in the diagram. Three main things should arise from this:

- The tractive system should be disabled, and stay disabled until reset by an non-driving operator (preventing DRIVE)
- A red clearly labeled BMS indicator light should turn on
- The tractive system indicator light should be turned on

Once the fault shutdown sequence has run, the state should default to IDLE and be unable to switch to DRIVE.

1.4 Important state changes

Switching from IDLE to Drive (or vice versa)

The car operator is able to switch from IDLE to DRIVE when the accelerator is pressed. Figure 1.2 at the end of this chapter details this process. When switching from IDLE to DRIVE, the BMS must ensure the following:

- The tractive system must not be disabled (indicating fault).
- All conditions must be within operable levels
- The shutdown signal must not have been triggered
- A fault must not have occurred

When switching from DRIVE to IDLE, there is less to be concerned with. This transfer can occur fairly seamlessly as long as the accelerator is not in usage and the user does not need more power from the BMS

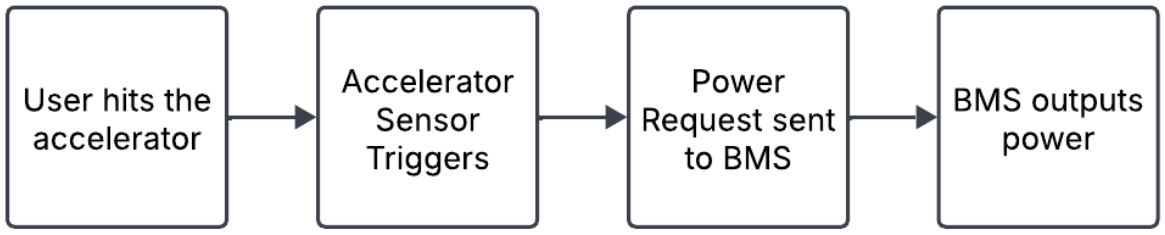

User hits the accelerator
Accelerator Sensor Triggers
Power Request sent to BMS
BMS outputs power

Figure 1.2: General process for acceleration

Charging the battery

Charging is a possible state for the BMS. The current car that FEB has does not support regenerative braking, therefore there is no charging state adjacent to DRIVE to display this. In order to charge, the battery must be unhatched from the car and manually plugged in to an external charger. Since this does not involve the general operation of the car, I have chosen to omit this detail in the diagram.

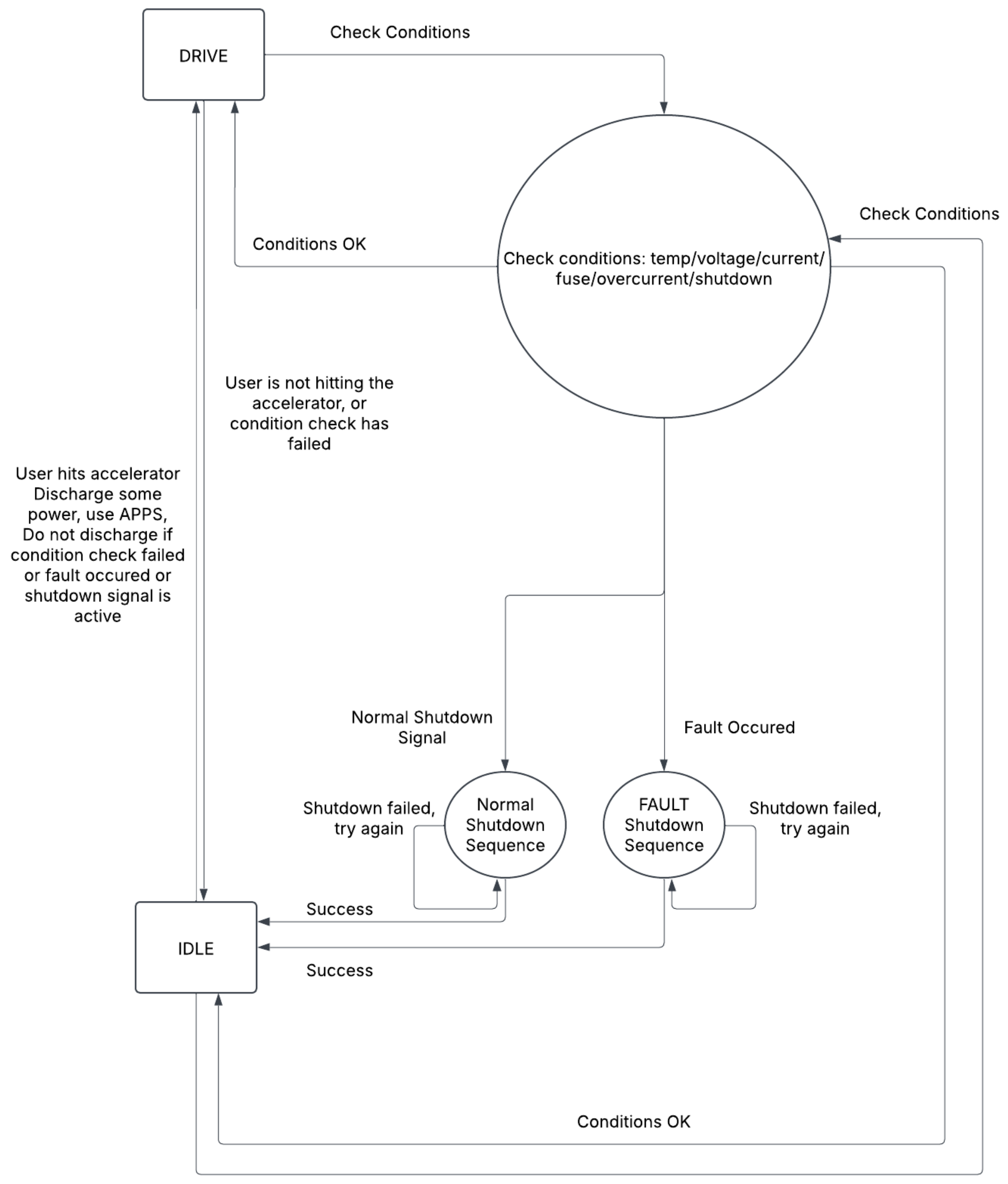

DRIVE
Check Conditions
Check conditions: temp/voltage/current/fuse/overcurrent/shutdown
Check Conditions
Conditions OK
User is not hitting the accelerator, or condition check has failed
User hits accelerator Discharge some power, use APPS, Do not discharge if condition check failed or fault occured or shutdown signal is active
Normal Shutdown Signal
Fault Occured
Shutdown failed, try again
Normal Shutdown Sequence
FAULT Shutdown Sequence
Shutdown failed, try again
Success
IDLE
Success
Conditions OK

Figure 1.3: BMS State Machine Diagram

Chapter 2

Code explanation

2.1 Introduction

The BMS and other firmware is primarily written in C, so to reflect that all the code for this project is written in C. The C99 version was used. This section details the libraries, functions, and process behind the written code. It's worth noting that if-else conditionals have been used to illustrate arrows in the original BMS Diagram and functions have mainly been used to illustrate states. In some cases, such as shutdown, a function has not been used. I have also made certain notes in the Github repository about possible future directions for the BMS demo project, such as multithreading and better methods for showing the BMS functions in real-time. While not numerous, I would encourage the reader, if interested, to consider and contribute if they would like.

2.2 Libraries

The following table details each library import and what it does.

Library	Description
stdbool.h	Allows for boolean variables to be programmed in
stdlib.h	C std. lib.; defines general purpose macros, variables, and functions
stdio.h	enables input/output commands
string.h	allows for simple string functions like strcmp()
windows.h	imports sleep() command if on a Windows machine
unistd.h	imports sleep() command if on a Linux machine

2.3 Helper functions

run_fault_sequence()

This function runs a few print statements to reflect actions that might take place when the fault sequence is run. Code is short and is pasted below.

```
/* Run fault sequence */
void run_fault_sequence() {
 printf("Running Fault Sequence:\n");
 printf("Disabling Tractive system\n");
 printf("Turning on BMS Indicator Light (RED)");
 printf("Turning on Tractive System Indicator Light");
}
```

turn_on_cooling_for_discharge()

This function takes in an (int) current value and calulates whether the current is within certain ranges in accordance with the battery specification. Depending on this value, it sets the cooling setting. The code is short and is pasted below:

```
/* Check current and set different kinds of cooling */
void turn_on_cooling_for_discharge(int current) {
 if (current <= 60) {
  printf("No cooling turned on (OFF)\n");
 } else if (current <= 120) {
  printf("Forced Air Cooling ON\n");
 } else if (current <= 180) {
  printf("10 sec. pulse, fuse limited.\n");
 }
}
```

check_conditions()

This function contains possibility the longest set of code and goes through multiple conditions to see if the BMS is operating as intended. The function returns a boolean value to reflect this. A copy of the code is below for your convenience:

```
/* Check conditions to see if each item is in acceptable range */
bool check_conditions(float temp,
                        float voltage,
                        float current,
                        bool fuse_OK,
                        bool overcurrent_OK) {

     printf("State: Checking all conditions of BMS \n");

     /* Check Temperature values */
     bool temp_OK = ((temp >= -20) && (temp <= 60));
     printf("Temperature: %s\n", temp_OK ? "OK" : "NOT OK");

     /* Check Volage value */
     bool voltage_OK = ((voltage >= 2.5) && (voltage <= 4.2));
     printf("Voltage: %s\n", voltage_OK ? "OK" : "NOT OK");

     /* Check current value */
     bool current_OK = (current <= 180);
     printf("Current: %s\n", current_OK ? "OK" : "NOT OK");
        turn_on_cooling_for_discharge(current);

     /* Calculate if all conditions OK & fault if not. */
     bool ok = (temp_OK &&
                voltage_OK &&
                current_OK &&
                fuse_OK &&
                overcurrent_OK);
     if (!ok) {
      printf("Not all conditions OK: FAULT occured.\n");
      return false;
     }

     // All Checks succeeded, return true
     printf("All checked passed.\n");
     return true;
}
```

2.4 Main function

The main function is composed of five sections: file reading and variable allocation, file printing, actual BMS operation, and memory deallocation. I've omitted the file reading and memory deallocation for this section for better readability. This function controls the bulk of the main transitions between the DRIVE and IDLE state and also triggers other sub-states, such as the fault sequence if needed. A segment of this is shown below.

```
/* Read the provided TEXT file to get in test inputs */
 /* Implementation omitted */

 // Print all the file information;
 /* Implementation Omitted */

 /* LOOP the entire system forever until the USER quits the program

  START either at DRIVE or IDLE
 */
 printf("/*************************/\n");

 while (true) {
  printf("/*************************/\n");
  printf("Current State: %s\n", curr_state);
  printf("/*************************/\n");
  #ifdef defined(SLEEP)
  sleep(2); // Sleep so that we can witness changes across time.
  #endif

  if (strcmp(curr_state, "DRIVE") == 0) {
   bool ok = check_conditions(*temp,
                              *voltage,
                              *current,
                              *fuse_OK,
                              *overcurrent_OK);

   // if shutdown signal is hit, then switch the state.
   // execute the correct type of shutdown
   if (*shutdown_signal) {
    printf("Shutdown signal triggered. Changing State to IDLE.\n");
    strcpy(curr_state, "IDLE"); // switch state to IDLE
   }

   // some fault occured, run fault sequence
   if (!ok) {
    run_fault_sequence();
    strcpy(curr_state, "IDLE"); // switch state to IDLE
   }

   // if accelerator is NOT in use, then switch to IDLE
   if (!*accel_in_use) {
    strcpy(curr_state, "IDLE"); // switch state to IDLE
   }
  }

  else if (strcmp(curr_state, "IDLE") == 0) {
   bool ok = check_conditions(*temp, *voltage, *current, *fuse_OK, *overcurrent_OK);

   // if accelerator IS IN USE, then switch to DRIVE
   if (*accel_in_use) {
    strcpy(curr_state, "DRIVE"); // switch state to DRIVE
   }

   // run the fault sequence if condition check failed. Do not switch to drive
   if (!ok) {
    run_fault_sequence();
   }
  }
 }

 // free all allocated variables (if needed)
 /* Implementation omitted */

 return 0;
```

This follows the general flow proposed by the BMS diagram.

2.5 Testing

Testing occurs through text files labeled input.txt. See example-input.txt and input.txt in the Github repository to get a good example of this. The file example-input.txt has been copied below for your convenience:

example-input.txt
Curr_state
Temperature
Voltage
Current
Fuse_OK
Overcurrent_OK
Shutdown_OK
Accelerator_IN_USE

Each line can be replaced with certain inputs to trigger different one-passes through the BMS State Machine. The first four values are numerical values and the program will interpret them as floats respectively. Fuse, overcurrent, and shutdown should use the value OK or NOT OK to show their state and are interpreted as booleans respectively. Accelerator should be put as TRUE or FALSE and is also interpreted as a boolean. I have purposely not used a while loop to iterate over this file to prevent the creation of a holder array and save space in the heap as I do so. I understand in firmware, memory, among other items, might be critical to keep track of.

A segment of this code has been pasted below:

```
...
/* Read the provided TEXT file to get in test inputs */
 FILE* file = fopen("input.txt", "r");
 if (file == NULL) {
  printf("Error: File Not Found. \n");
  return 1;
 }

// Load in items to the HEAP from FILE.
char *curr_state = malloc(sizeof(char) * 20);
fgets(curr_state, sizeof(curr_state), file);
curr_state[strcspn(curr_state, "\n")] = '\0';  // Remove newline

char buffer[20];

// Read temperature
read_line_into_buffer(buffer, file);
float *temp = malloc(sizeof(float));
*temp = atof(buffer);

// Read voltage
read_line_into_buffer(buffer, file);
float *voltage = malloc(sizeof(float));
*voltage = atof(buffer);

// Read current
read_line_into_buffer(buffer, file);
float *current = malloc(sizeof(float));
*current = atof(buffer);

// Read fuse_OK
read_line_into_buffer(buffer, file);
```

```
    bool *fuse_OK = malloc(sizeof(bool));
    *fuse_OK = (strcmp(buffer, "OK") == 0);

    // Read overcurrent_OK
    read_line_into_buffer(buffer, file);
    bool *overcurrent_OK = malloc(sizeof(bool));
    *overcurrent_OK = (strcmp(buffer, "OK") == 0);

    // Read shutdown_signal
    read_line_into_buffer(buffer, file);
    bool *shutdown_signal = malloc(sizeof(bool));
    *shutdown_signal = (strcmp(buffer, "TRUE") == 0);

    // Read shutdown_signal
    read_line_into_buffer(buffer, file);
    bool *accel_in_use = malloc(sizeof(bool));
    *accel_in_use = (strcmp(buffer, "TRUE") == 0);
    ...
```

2.6 Etc

Links

The Github Respository along with all the code can be found at the following link:
https://github.com/devenbibikar/bms-feb

Disclaimer

The code does not contain any support for charging states. This is not included in the diagram and therefore is not reflected in the code. Refer to Chapter 1 for more information on this. Instructions have been omitted in this document and can be found directly in the repository.

Possible Future Directions

A clear pitfall posed by this code is that it is only capable of doing a one-pass through the program and does not have fault tolerance for malicious files. This can be amended by creating multiple threads that could do the following tasks: query the file to see if there are any changes that need to be updated, have a thread that updates the values from the file, and have a thread that broadcast s to the main thread when a state change could take place.

For this, a thread-safer language like Rust is recommended. Fault tolerance for files can be implemented by checking the file formatting every time file read takes place.

Chapter 3

Final Words

3.1 Lessons Learned

When I first applied to this team, I didn't know what a BMS was. I had never worked with batteries, and really never thought of them either. I also never thought about how the formula electric car actually functions. I think that in a very short span of time I've managed to learn a lot and for that I'm very grateful. It was fascianting to see how electric cars, notably the formula electric car, differed from a standard gas vehicles.

In truth, before this, the most complicated state diagrams I came across were the ones in *CS61C*, most of which would only contain 2 or 3 simple states and very little design from me. Creating a state machine that's accurate enough to drastically simplify code is an interesting workflow I've never done, and I think that's an experience worth noting. I really liked the freedom in design in the coding section because it gave me a lot of avenues to think of how to organize my code and how work could be improved. I thought it was a good exercise in coding creativity.

3.2 Acknowledgements

I would like to thank all the Formula Electric members for their advice and mentorship as I designed this state machine and learned about the BMS. It's been really fun getting to know all of you, and I wish you all the best.

3.3 Document Credits

This Latex template is based on the UC Berkeley Graduate Office's PHD Thesis Template.