Formula Student Team Delft
Delft University of Technology

Battery Management System (BMS)

Federico Fiorini
federico.fiorini@outlook.it *+39 3456181618*

TU Delft Delft University of Technology

I'd like to thank everyone in the Formula Student Team Delft that helped building the DUT19 racecar, in particular all the members of the Electrics department and the Chief Electric for the beautiful months spent together.

Version 1.1.0, March 2020

Contents

Chapter 1

Introduction

This document explains in a quite extensive way the code for the **Battery Management System (BMS)** of the DUT19 race car, which was written from scratch last year.
This small reference guide focuses on explaining the basics of the BMS and the Low Voltage system operations, without going too deep into the hardware details, as the overall PCB design has been developed by an Hardware Engineer from the team, and I don't have permission to share its design details. The code description starts in the next chapter, after a short introduction on the most important hardware components and a high-level description of the code structure and operations.
In future versions of this guide I will include some general guidelines on embedded testing, including indications on how the testing phase was tackled for the BMS. At the moment, there is no indication on testing and it might take some time before getting released to the public.

Chapter 2

Battery Management System (BMS)

The **Battery Management System**, or **BMS** in short, is the printed circuit board responsible for managing the **Low Voltage Battery**, or **LVB**, and ensuring its safe operations.
It's a relatively new addition to the codebase and the car's electronics, as before DUT19 this component was always bought "off-the-shelf" with no need for the team to design electronics parts or to write custom software. Rules changes from Formula Student Germany required the team to change this approach and design, build and test its own electronics and embedded software for the Low Voltage Battery monitoring system.

2.1 Overview

The LVB controls all the low-voltage electronics in the car, including all the PCB's, the cooling fans and water pumps, the DRS servo-motors and some safety devices required by the rules (such as the TSAL and the IMD, which I'm not going to cover in this small reference guide).
It's essential, when writing software for a component managing batteries or power electronics, to be sure that its behaviour is well-defined and is controlled at any given time, to ensure the safety of the car, the drivers and anyone around. Imagine, for example, what would happen if suddenly there's a short circuit in the battery and no countermeasure is designed and taken to mitigate that.

The LVB-BMS system has a different operational mode with respect to the High Voltage (HV) system of the car, as it's way smaller and it requires different / less stringent safety measures.
The LVB itself consists of 7 cells in series (7s), providing an operational voltage of between 21V and 29.4V. It mostly operates at around 24.5V when the car is running.
The operating range is given by the operational voltage of the battery cells themselves, which are required to operate between 3.0V and 4.2V (any value below 3.0V is referred to as *"undervoltage"*, or *"UV"*, and any value above 4.2V is instead mentioned as *"overvoltage"*, or *"OV"*, in the remainder of this guide).

The safety mechanisms are enclosed in the PCB itself, as well as the switching mechanisms that allow the current to flow from (when the car is powered on) or to (when the LVB is charging) the battery: this configuration and simplicity makes this system a lot smaller than the accumulator, which requires a whole separate compartment for the safety components and switching mechanisms, due to safety reasons.

For the sake of knowledge, the DUT19 LVB can provide current up to 18A at maximum power drawn, with an expected battery duration of 30 to 40 minutes when fully charged (which is sufficient, under normal conditions, to finish an Endurance race without discharging the battery completely).
This expected working time is influenced by the cells' lifecycle and environmental factors, so it's possible for it to decrease over time and under different weather conditions.

2.2 Hardware

This guide will not provide design or implementation guidelines or references on the design of the LVB (which is property of the Powertrain Department) nor on the specifics of the PCB design (which is property of the Electronics Engineer who designed it).
On the other hand, it will provide some little explanation on "off-the-shelf" components and on how they integrate in the overall software design.

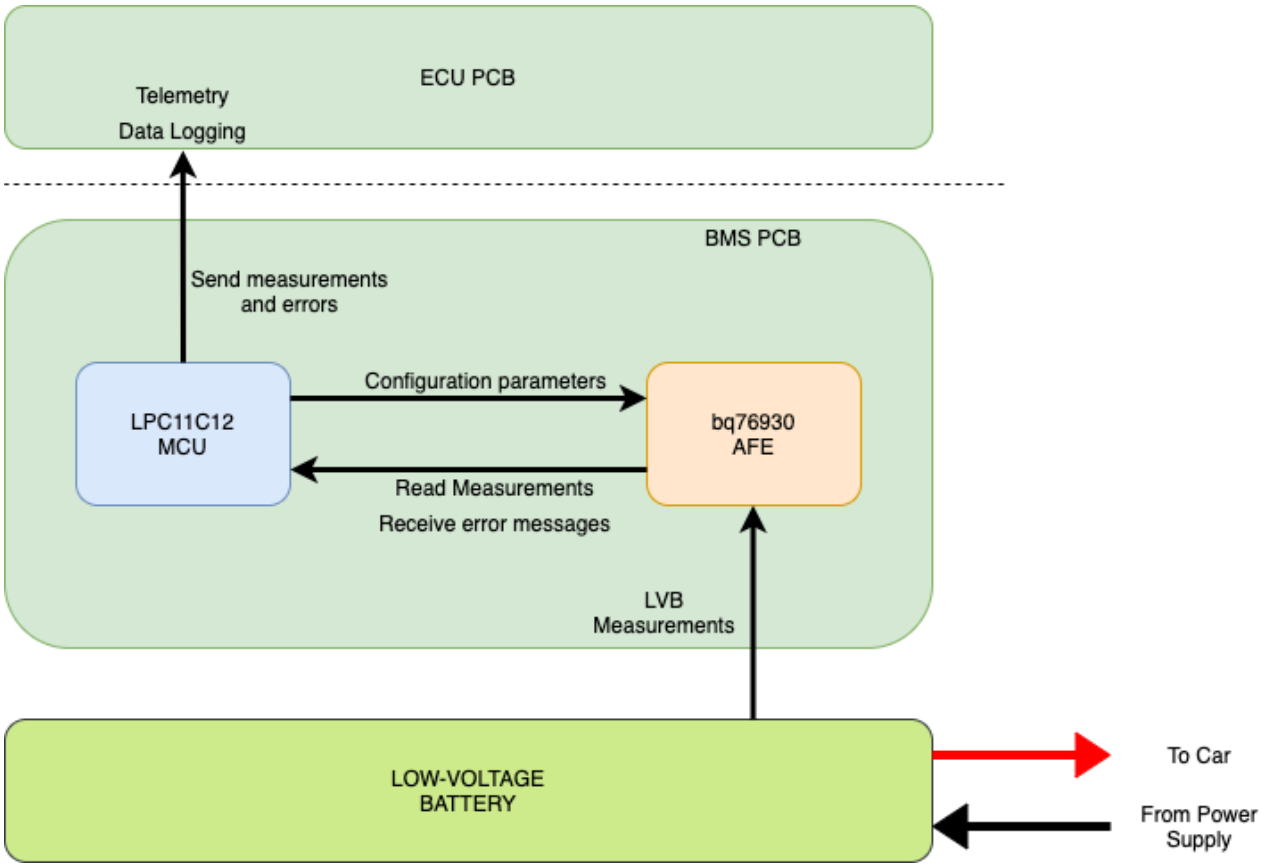

Figure 2.1: BMS Hardware Configuration

As you can see from the picture above, there are two different controllers which work together to ensure the

correct behaviour and safety of the LV system (and the car itself) at any time: the *Analog Front End* and a *Microcontroller*.

The **Analog Front End** (or **AFE**, for short) is a small monitoring device produced by Texas Instrument, meant specifically for monitoring of Li-Ion batteries and systems. This chip is directly connected to the battery cells and it directly monitors its behaviour, such as individual cells voltages and pack voltage, and it can be directly configured externally to achieve the desired configuration and level of control.
We don't have direct access to the AFE's source code, but it's possible to write configuration values to and read measurements from its registries, which are extensively described in the datasheet.
Some of the configuration options include, among the others, setting the OV and UV thresholds, the *Over Current* and *Short-Circuit* (*OCD* , *SCD*) protection thresholds, and the delays after which such errors will trigger. It also offers the option to individually balance cell voltages during (dis-)charge operations and to set measurements frequency.
As said before, the AFE can provide a wide range of measurements: each individual cell voltage, the battery (pack) voltage, the state of charge of the battery and its temperature. It also gives the option to read detailed error syndromes by retrieving a register's value, and to quickly inform the companion controller (in this case the MCU) of an error occurring by simply pulling a pin high.
We don't use the built-in thermistors in our design, since, by regulations, we need to perform more measurements than the two provided by the AFE by default.

Given these rules changes for 2019, we decided to enhance the monitoring capabilities of the AFE with the **LPC11C12 MCU**, produced by NXP and comprising an ARM-Cortex M0 processor and several analog and digital peripherals to support its operations.
This chip has been chosen for its very-low power consumption, its simplicity, the possibility to embed it directly onto the PCB with little to no struggle and the possibility to enable a so-called "Deep Sleep Mode", which further reduces power consumption when the BMS is not needed inside the car.
The MCU is responsible for collecting all measurements from the LVB, reading the appropriate registers in the AFE, and send them to other components in the car (namely, the Electronics Control Unit, or "ECU", responsible for data logging and telemetry, among other duties). It's also responsible for writing the correct configuration values to the AFE if required, and listening for any error syndrome or issue coming from the LVB and take the appropriate (custom) countermeasures.
It's also used to measure the cells temperatures using three different thermistors, directly connected to the LVB on one side and to the MCU's ADC lines on the other: with this enhancement, the LVB is fully monitored at any given time, meeting the safety requirements.

2.3 Operations and Safety

This section covers the security features of the BMS PCB and the operational behaviour from a high-level perspective.
The LVB is enabled (i.e. connected to the rest of the car) or disabled using two simple MOSFETs connected to the AFE, which decides at any time in which configuration they can stay. Their behaviour is strictly influenced by the measurements coming from the battery, meaning that, in emergency situations, the two MOSFETs will always be open, so the LVB is disconnected from the rest of the car. The behaviour of the MOSFETs is further regulated by the MCU, which cooperates with the AFE to ensure safe operations.
Namely, the two MOSFETs are called

- **DSG (Discharge)**: when closed, the LVB powers the whole car's LV systems

- **CHG (Charge)**: when closed, the LVB gets charged from an external power supply

When the car is shut down (i.e. the *LV Master Switch* is open), the DSG MOSFET is also open and thus the LVB is disconnected. It stays disconnected as long as the Master Switch is open.
Upon switching the Master Switch to the closed position, the LVB is connected to the rest of the car and the BMS is also starting its operations, which will then be described with some more details in the code section.

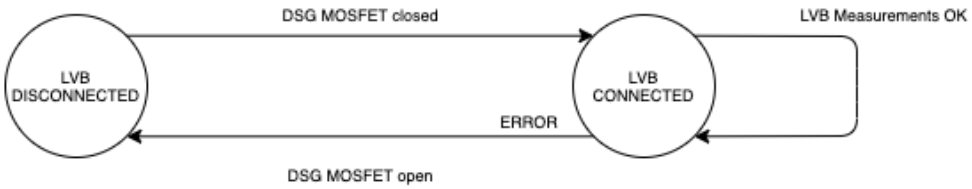

Figure 2.2: Overview of LVB states

As you can see from the picture above, the DSG MOSFET is closed upon switching the Master Switch and the current starts flowing from the LVB to the rest of the LV system. The BMS constantly monitors the LVB measurements, coming from both the AFE and the MCU's ADC lines, to ensure that the battery behaves correctly and according to the safety regulations.
In case an error is triggered, the LVB is promptly disconnected from the rest of the LV system and the car is then shut down again. We will see that the error syndromes can also be recoverable, but as a first step the car needs to be safe.

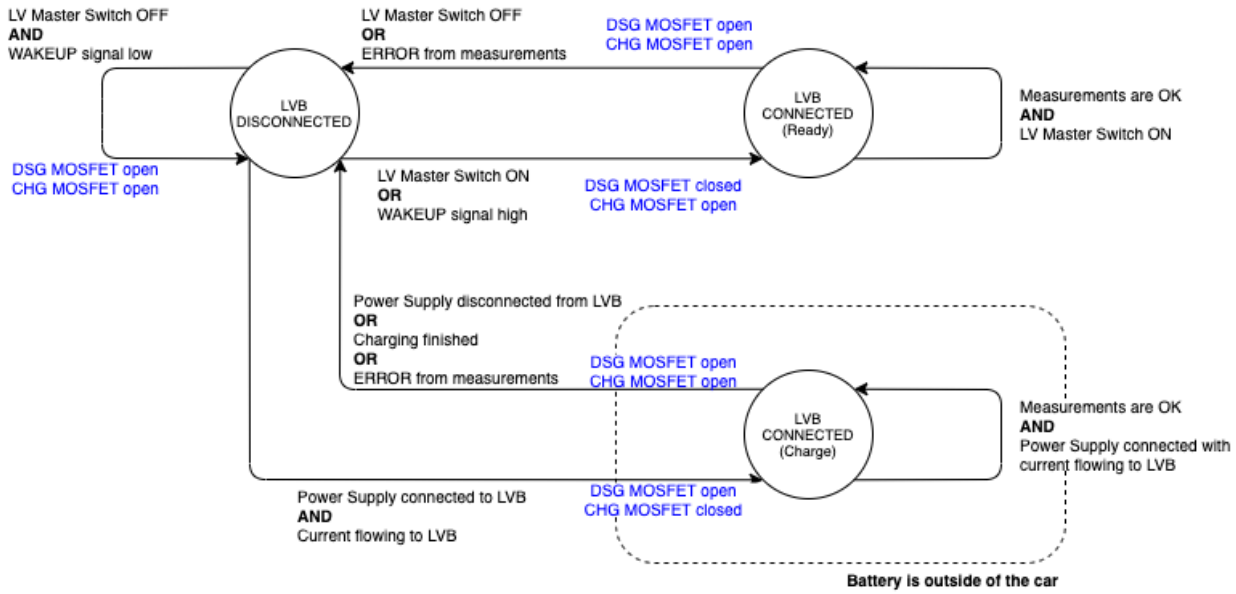

Figure 2.3: LVB Operations in detail

Figure 2.3 explains the LVB procedures with a higher level of detail: it's possible to see that the LVB is connected to the system under two different states, Ready and Charge. The former refers to the set of states in which the LVB powers the other components in the car, so the current flows from the battery; the latter refers to the Charging procedure, where the battery is stored in a safe environment (outside of the car) and connected to a power supply, so the current flows towards the battery.

During all states the battery is constantly monitored by the hardware subsystem, in particular the AFE. This component is constantly measuring data coming from the battery, and if something happens when the battery is disconnected from the car it quickly triggers an interrupt onto the MCU, which then "wakes up" and react accordingly. More on this in the code section of this guide.
Under normal conditions the LVB stays connected until it reaches its minimum voltage, which is driven by the lowest cell of the pack: if a single battery cells drops below 3.0V, the whole pack signals an UV condition and the battery is disconnected. Once it happens, the LVB needs to be re-charged before being re-used.
Sometimes it's good practice to check the status of the cells, as the voltage of each of them might differ significantly from the others, causing the battery life-span to decrease consistently.

A solution to this problem is called "Balancing", which will be discussed later. If balancing doesn't solve the problem, the battery cell can eventually be swapped with a new one (if available).

As I mentioned earlier in the guide, this system is capable of generating quite high currents during driving conditions: at startup, the current drain usually lies between 3A and 4.5A, mainly caused by the powering up of the cooling fans and the water pumps. The current flowing from the battery starts to increase as soon as the car starts driving, and it gets up to 18A during maximum performance (i.e. when the fans and pumps are at maximum power to maintain an acceptable temperature range for the motors, the brakes and other HV components).

During charging operations, the recommended current from the power supply has to be around *10A maximum*, to avoid electrical issues with the battery cells. Charging, as it can be seen from Figure 2.3, is activated whenever the power supply is attached to the LVB (which can be performed only if the battery is outside of the car) and the CHG MOSFET is closed: this MOSFET is closed automatically by the BMS (in particular, the MCU) upon sensing a current flowing in the opposite direction than the normal one.
The LVB will be charging until the pack reaches its maximum allowed voltage (which is set in the code to be 29V, to maintain some "safe threshold" with respect to the maximum electrical rating of the cells), or as soon as one cell goes into overvoltage, thus exceeding 4.2V.

Some other safety components on the PCB are the **fuses**: there are several of them placed in critical parts of the BMS PCB, such as the power lines from and to the LVB, the inputs of the AFE and MCU (Input Protection) and the ground lines. In this way, the PCB and its components are always protected from very high currents which may generate when the LVB is not functioning properly. These fuses also protect the rest of the LV system, and together with other safety systems in the car ensure the safety of the car at any given time, to protect the drivers and the people working around it.
Fuses are also a very good way to determine whether the PCB is still functioning properly, or whether the LVB is behaving correctly.

During charging and standard operations, it's possible to *balance individual cells*: this procedure, engaged using a button placed on the PCB, lets some battery cells discharge independently from the others. In this way, all cells will eventually align to a certain voltage level, ensuring the maximum capacity for the battery.
There are some limitations on this procedure given by the AFE datasheet and the cells themselves, in particular it's stated that two adjacent cells can't balance at the same time, in order to avoid electrical issues.

Finally, it's important to remember that the LVB has to be stored in an appropriate fire-retardant and fire-resistant metal container, to ensure the safety of the workshop and the people working there.

2.4 Code

The entire codebase of the BMS is contained in the code folder of this project, and this section will provide a high-level overview of the main components as well as how the codebase works.
The entire BMS project is *self-contained*, meaning that it has no dependencies on other parts of the DUT19 car's embedded software: this is a direct consequence of using an ARM-Cortex M0 powered MCU, whereas the other PCB's use a more advanced and complex ARM-Cortex M3 processor.

The main characteristic of the BMS software is the **absence of a Real-Time Operating System (RTOS)** in the codebase, since its operations and state machine can be encapsulated in a simple while loop. Moreover, given the simplicity of the BMS's operations, building a RTOS-driven codebase would have been useless and too much over-engineered.
Other systems in the car use the RTOS scheduling capabilities to perform more complex and customised behaviours by enabling multi-threading and several memory-consuming and I/O-enabled operations. The BMS, to put it in simple terms, just retrieves measurements from the LVB and sends them over to other PCB's, where they can be logged and monitored even further.
This data communication, furthermore, *is not strictly necessary for the car to drive*: the BMS acts as sole responsible for taking countermeasures in case the LVB presents issues, and thus it only follows its behaviour at any given time. It has no knowledge of the rest of the car's status, and just performs simple tasks to ensure the LV system works safely at all times.

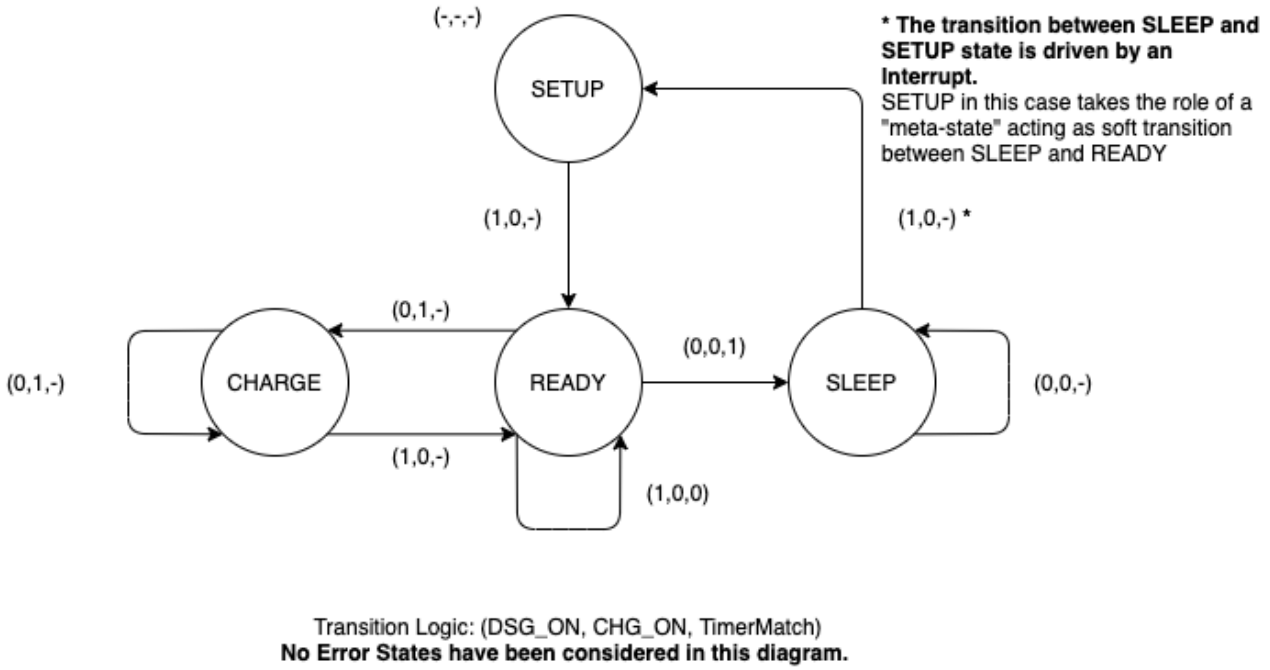

Figure 2.4: Brief description of the BMS State Machine

Figure 2.4 shows a high-level and not full state machine of the BMS, which doesn't include error states and balancing for ease of understanding.
The four main states in which the BMS can be are, as you can see from the picture:

SETUP The initial state, responsible for initialising all peripherals and modules;

READY This is the state in which the BMS measures all data from the LVB and sends it to the other systems in the car, using its CAN lines;

CHARGE This is the state in which the LVB is charging and placed outside of the car. The measurements still take place, but the MOSFET configuration differs from the READY state;

SLEEP This is the ultra-low-power consumption state, which is reached after the LVB is disconnected from the car and stored in a safe container.

As you can see, the SETUP state can be reached in two different moments: at startup and when the BMS "wakes up" from the SLEEP state. In the latter option, as it's shown in the picture, the SETUP state acts as a *soft transition* between the SLEEP and READY states.

Once the battery is disconnected from the car and placed somewhere else temporarily (either on a table or in the safe container), both DSG and CHG MOSFETs are open: the moment the MCU doesn't sense any current flowing in either direction, it starts a hardware timer which, upon reaching a pre-determined value (that can

be approximated to 5 minutes), starts the procedure to put the BMS into **"Deep Sleep" mode**.
This mode, referred in the state machine as the SLEEP state, is a very-low-power consumption mode typical of the LPC11Cxx processors family: in this mode, the processor goes to an idle state and all the peripheral are shut down, with the exception of a few GPIO pins which can be used to wake the MCU up again when needed. Three of these pins are currently used in the BMS to wake up the chip:

ALERT Directly connected to the AFE, which still measures the LVB. It goes high when the AFE senses an issue with the battery, alerting the MCU to take countermeasures;

WAKEUP Connected to a button on the PCB, it's used for manual wakeup of the MCU;

SENSE This pin is directly connected to the power lines, and it measures the current flowing to or from the battery: this way, whenever a power supply is connected to the LVB or the LVMS is turned on, the MCU wakes up and resumes its operations.

Once the GPIO pins go high, an interrupt is triggered within the MCU, which in turn calls a handler to act on the processor: first of all it powers up all the peripherals and drivers, then resumes code execution from the beginning of the while loop. This soft transition was needed to let the processor be able to fully resume its operations without any issue.

Let's now dig deep into the codebase, with a bit of explanation on its structure and the code contained in it.

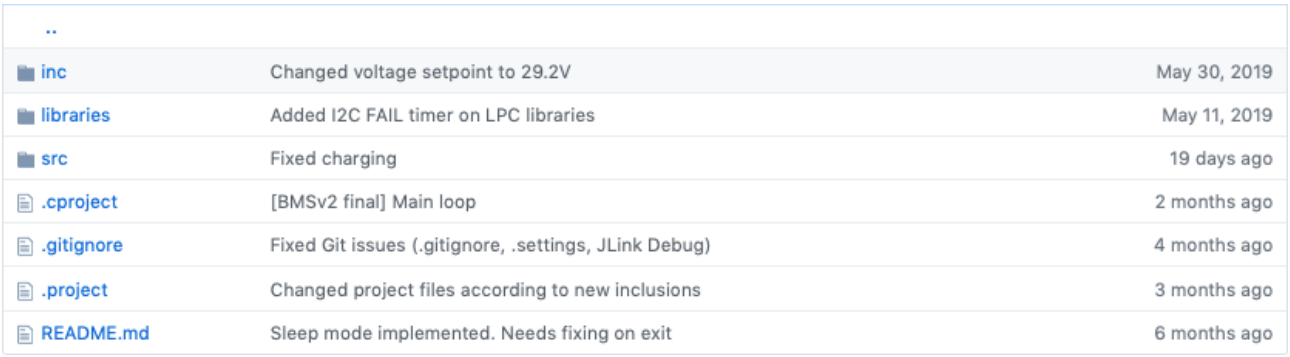

Figure 2.5: BMS Code Structure

As you can see, the source code is divided into three main folders, plus some additional files used by GitHub (which is the source control tool used by the Formula Student Team Delft for the entire car's codebase).

inc Contains all the header files;

libraries Contains the third-party-libraries and support code, such as the processor's source code and the SEGGER RTT libraries used for debugging;

src Contains the source files with the actual implementation, and the main loop.

The entire codebase adheres to the principles of Software Engineering, where the code is divided into headers/source files to separate the declaration and the actual implementation of classes and methods, and the functionalities have been separated accordingly to avoid monolithic architectures or "God classes".
Each of the following subsections describes in detail each main component of the BMS codebase.

2.4.1 Peripheral Drivers

A big portion of the codebase comprises the peripheral drivers, which act as an "abstraction layer" between the MCU source code and the application level.

The drivers are built on top of the source code for the MCU, provided by NXP, and include the GPIO peripheral pins, the Analog-to-Digital Converter (ADC), the I2C, CAN and UART communication peripherals and interfaces, and the Hardware Timers. UART is implemented but never used, as the idea of having a direct connection between a PC and the BMS (in order to use a monitoring application when charging or checking the state of the LVB) was later discarded due to time constraints.

All the drivers included in this codebase have a similar structure, as they all have a method to correctly initialise the peripheral (according to requirements and guidelines from NXP itself) and several functionalities to use the peripherals: in case of the communication drivers, they all have methods to send or receive data (as well as transceive functions, which combine both send and receive).

The GPIO peripheral is not actually initialised within the GPIO driver, but rather in the Pins section of the code. This decision was made to aggregate all functionalities related to the GPIO pins in two files (header and source) only.

The only "peculiar" driver is the ADC: this peripheral is also responsible for measuring the temperature of the LVB cells and the current flowing from and to the LVB (using a sensing point on the power lines on the PCB). Both measurements use the built-in ADC functionalities to read data from a particular channel, which is one of the 8 analog input pins of the converter.

The **current measurement** needs to take into account for an amplification stage, for which an OPAMP has been used, which helps getting a more accurate current measurement than simply sensing the line without it. This stage was mandatory since the digital readout is the voltage difference across a "sensing resistance": to avoid losing precision during the conversion, the whole reading has been amplified and shifted using the OPAMP threshold, enabling a more accurate digital conversion.
In order to correctly retrieve the measured current, it's required to use the following equation:

$$MeasuredCurrent = (((MeasuredVoltage * LSB) - OPAMPThreshold)/OPAMPGain)/Resistor$$

It's possible to find all the values (for LSB, Threshold, etc.) in the files, but it's important to keep in mind that all of them have been converted down to "μ - scale" to be sure to easily obtain a current in a scale of mA.
This has been necessary due to some inherited constraints given by the ARM-Cortex M0 processor: since it doesn't comprise a Floating-Point (FP) Unit, all operations had to be expressed as integer value, with a clear loss of precision in case of decimal values. Thus, I decided to limit the precision loss by having all the measurements expressed in terms of mA, mV, mΩ, etc.
This limitation influenced the temperature measurements as well.

The **temperature measurements**, in fact, are retrieved using an equation which, giving an input resistance (measured from the NTC thermistor and converted into digital value) and some characteristic values, it converts into a temperature value in degrees Celsius (the so-called *"Curve Fitting" Steinhart-Hart Equation*).
This equation heavily uses floating point operations, and the result itself would be expressed as a floating point value (in particular, double).
The ARM-Cortex M0 can of course allow to have single- or double-point arithmetics, but it applies something called *"Software-Defined Arithmetics"*, which heavily increases the memory usage of the generated assembly code (since it requires more fixed-point arithmetic instructions than FP-arithmetic).

Given the small amount of memory available, the approach taken was to use a **lookup table**, created using a MATLAB script, and incorporate it as a part of the codebase: this way, it has been possible to have a one-to-one mapping of the thermistor values and the correspondent temperature. It's worth noting that the equation inherently introduces errors in the final temperature measurements, since it tends to linearise the behaviour of a thermistor to a fixed curve.
Empirical tests showed how the drift between the actual value measured and the "calculated" one was negligible in the range of operating temperatures for the LVB (which is between 0°C and 60°C).

2.4.2 BQ76930 Driver

The **BQ76930 driver** takes its name by the Analog Front End device name, and it includes all the data structures and functions required to work with the AFE. It serves as an abstraction layer between the MCU and the AFE's internal registers and operations, and it uses the I2C communication driver to handle communications between the two devices.

Some of the functionalities included in this driver are reading data from and writing data to the AFE's internal registers, retrieve the cells and battery voltages and the battery's state of charge, and enable or disable balancing on specific cells.

The data structures include, among others, the AFE's registers, the ADC gain and offset (used in retrieving the measurements), some configuration values and flags. The whole set of operations and data structures can be found in the code folder, and it's worth remembering that the entire BQ76930 code has been developed using the device's datasheet, which gives a comprehensive view on all the internal details and the interfaces that the AFE provides to the outside.

The communication between the MCU and the AFE, as said before, is performed using I^2C and enhanced for resilience using *CRC8 error correction*, which enables a reliable communication between the devices and avoids misreadings. It also serves as a "double check" to ensure the connection between the two devices is working correctly.

Diving deep into the driver's code, it's possible to see the initialiser function, which enables the I^2C peripheral on the MCU and writes some configuration values to the AFE registers, such as OV, UV and OC protection thresholds, and initial value for cells balancing (which is disabled by default to avoid issues).
All the measurements and the configuration writes are based on two essential functions, namely

- `write_register`
- `read_register`

Which, as you may understand, perform read/write operations on the 8 bit registers: all measurements, then, are "built" concatenating several read operations and then apply some changes, according to the datasheet specifications (more on that later on).
As mentioned earlier, all read/write operations include CRC8 error correction in the payload, which is then discarded from the reading in case of measurements reading. The function creating the CRC8 code based on the data to be written is also included in this driver, and it's based on the basic version which can be found online from several sources. It's worth pointing out that not all families of the BQ769x0 devices support CRC8, so check it out before implementing the code (this information is included in the devices datasheet) to avoid communication issues.

Each cell's voltage is a 14 bit values stored in two adjacent registers of the AFE, so in order to retrieve it

it's required to perform two subsequent reads from the registers, then build the final 16-bit (for simplicity) measurement using bit-shift and bitwise operators.
The voltage data then needs to be modified using the ADC gain and offset parameters provided by the AFE, to have the final and actual cell voltage. The equation to be used, included in the datasheet, is

$$Voltage = (ADCout * Gain)/1000 + Offset$$

The factor 1000 has been included because, to avoid problems with floating point values, all readings have been transformed into mV: given this requirement, the ADC Gain is thus expressed in terms of μV/LSB, while the Offset is in mV (by default). The final measurements are expressed in mV.

The LVB Pack Voltage, on the other hand, requires a slightly different equation: this 16-bit value is yes retrieved using the same reading and bit-shifting techniques as the individual cells voltages, but the equation to use is

$$LVBvoltage = (ADCout * Gain * 4)/1000 + (Offset * \#cells)$$

The equation is provided by the datasheet, with the proper explanation of all the reasoning behind these values. The 1000 factor has been included for the same reason as above.

The cells voltages measurements are retrieved and calculated all at once, by simply calling the read_cellvoltages function: this function calls the individual reading functions for all the cells in the LVB, and it automatically calculates the minimum, maximum and average cell voltage of the whole battery, which would then be sent out for logging (and performance) purposes.

The AFE also provides another measurement, which is the State of Charge of the battery: it refers to the level of charge of the LVB with respect to its capacity. It can be retrieved and calculated using the correct AFE registers and the following formula, which can be found in the datasheet as well:

$$SoC = [16bit - 2's - complement] * (8,44\mu V/LSB)$$

The BQ76930 driver is also responsible for the **cells balancing**: under certain conditions (and only if manually initiated) it's possible to discharge certain cells to align all the cell voltages together, thus ensuring the LVB can last longer during its functioning (and in order to prevent excessive degradation on certain cells only).
The balancing procedure itself consists of different phases, each of which has to be strictly regulated to ensure it's safe at any given time:

- Balancing Enabling
- Balancing Condition Check
- Adjacency Check
- Balancing
- Balancing Stop Condition Check

These "sub-routines" are automated and the first one starts immediately after the correspondent button is pressed on the PCB. It can be pressed only when the BMS is placed outside of the car, to ensure the balancing discharge doesn't get affected by (or it doesn't cause any issue to) the LVB normal discharging.
The *balancing condition* is checked every 10-15ms (roughly the time for the main loop to complete) and ensures that the cells to be balanced are not the ones with the lowest voltage and that the number of balancing cells

doesn't exceed the maximum allowed. The *balancing procedure* aligns all cells together to a target voltage by discharging cells, so it comes naturally that all cells are aligned to the lowest one. The maximum number of balancing cells is determined by the battery cells themselves (and the optimal electrical requirements of the PCB).
The *adjacency check* ensures that, at any given time, no adjacent cells (as intended connected to the AFE, so not to be confused with the LVB) should discharge simultaneously through the balancing circuitry: this check prevents electrical issues on the AFE and PCB.
Balancing is enabled for a specific cell by writing a "1" on the correspondent register bit (there's two AFE registers which control balancing), and then the AFE does the rest.
The *stop condition* is a procedure called at the end of every loop cycle to ensure that, upon reaching the desired goal, "0"s are written on all bits in the balancing registers and, consequently, the balancing procedure is stopped: this allows for an automated functioning of the whole procedure, allowing more safety.
The stop condition checks whether all cells voltages lie in a predefined range, expressed in terms of minimum voltage plus a threshold. The threshold value has been decided at random and it's included in the configuration file.

2.4.3 Pins

The **pins** folder contains the definitions and the source code for the GPIO pins, which are defined (and named in a more useful and meaningful way than the MCU's source code naming convention) and initialised according to their use.
It's worth pointing out that, despite ADC and communication pins could be used as general-purpose inputs and outputs (and they belong to the same "ports"), *only the pure GPIO pins* are defined and initialised here: the I^2C communication pins, for example, are defined in the I^2C peripheral driver.

Among the pins there are the LEDs, which provide immediate and visual representation of the state of the LVB+BMS system. The LEDs have been encoded in such a way that every single state (especially error states) are mapped with a unique sequence of LEDs, so engineers have an easier time understanding what is happening in the BMS, or during monitoring (for instance, when charging the LVB).
The LED encoder document has been included as an appendix to this document.

2.4.4 State

The **BMS state** code is one of the most important components of the codebase, as it regulates the code behaviour and it provides real-time data on the status of the system.

The state header file defines the various states in which the BMS can be, especially the error states: in this way, it's possible to take the appropriate countermeasures based on what is happening to the LVB.
In the file there's also a definition of the so-called *"legal transitions"*, which is the set of state transitions that have been defined for the BMS during its normal operations. This term doesn't apply to error syndromes, as whenever an error occurs the overall state of the component immediately turns into the specific error state. It's important to notice that, according to the implementation (and driven by the safety regulations), all error are to be considered unrecoverable: this means that, whenever an error occurs, the LVB is immediately shut down and disconnected from the rest of the car, to avoid incurring in other (more severe) errors and issues.
It has to be said that, in case of transient errors, after the error syndrome has been checked and counteracted, the battery can still be re-connected to the system. Some errors, in the end, don't immediately disconnect the LVB, but give a "safe window" of a few seconds before taking countermeasures, to give the time to the drivers

to safely react and, for instance, park the car on the side of the track.

Most of the error syndromes are defined by the AFE readings, such as over-/under-voltage on particular cells, and the main state function just checks, at every loop cycle, that all the measurements from the AFE are okay, and that the temperatures from the ADC are in the correct operational range.

The AFE maintains a register, called `Status`, which contains a series of possible errors that might happen in the LVB: whenever the MCU reads a "1" in such register, it immediately converts the reading into a known state, then it lights up the LEDs according to the syndrome to signal the error to the engineers. It goes without mention that it also opens the MOSFETs, which effectively disconnect the battery.
Some errors, like the `OVRD_ALERT` and the `AFE_FAULT` are directly related to the status of the AFE itself, where the former signals that the ALERT pin of the AFE is high, signalling an error has happened, and the latter refers to a malfunctioning in the AFE. The ALERT pin is used by the AFE to signal an error of any sort has happened, and needs to be manually pulled down by the companion processor before continuing the operations.

The state code is also responsible for the **Deep Sleep mode**, and for waking up from it: this ultra-low-power consumption mode is triggered using a specific sequence of commands, that are described in detail in the LPC11Cxx datasheet.
This mode disables all peripherals (except for some interruptable pins used to wake up the MCU), enables and uses the internal IRC oscillator as a clock source, and simply puts the processor in a busy "wait for interrupt" routine. This state is triggered whenever the LVB is disconnected from the rest of the car (with both CHG and DSG MOSFETs open) for more than a certain amount of time, which has been set to be around 10 minutes. This way, the battery can be safely stored in the appropriate container, while the AFE is still monitoring it for added safety, and there's no significant current drain from the battery.
According to calculations, it should remain in this state for up to 6 months, if triggered at full charge.

In order to wake up from the Deep Sleep mode, it's required to trigger one of the wake up pins. There are three ways the BMS can be waken up:

- Pressing the WAKEUP button on the PCB
- Connecting the system to the car (and closing the LVMS)
- Connecting the system to the power supply (and triggering the charging threshold)

In each way, the transition between Deep Sleep and Ready states is almost immediate, as it's regulated by a specific *interrupt handler*: this handler re-enables and updates the processor's clock (set to be the System PLL), powers up all the peripherals, and initialises some relevant variables to their initial state. Once it's completed, the code resumes from the beginning of the main loop.
It's essential to reset the state of the interruptable pins and the interrupt vector, to ensure not to remain stuck in the handler's operations.

2.4.5 Main Loop

The main loop is just a big while loop that runs as long as the PCB is powered, right after some initialisation routines.
It's the main entry point of the code, and it regulates the BMS behaviour at any given time. It starts, as said earlier, with a call to all initialisation functions (MCU's peripherals and AFE) and with a global variables initialisation.

In addition, it performs two consecutive readings of the AFE's state, in order to get rid of transient errors before starting the normal operations. Once this step is completed, the while loop starts and, in case everything's okay with the LVB and the other systems on the PCB, the BMS operates in its ready state.
During its normal operations, the main loop is responsible for

- Measure the LVB (reading from AFE and from ADC)
- Retrieve the status
- Check for balancing enabling triggers
- Send data to the ECU (using the CAN loom)
- Check whether the two MOSFETs are open and, if so, initialise Deep Sleep transition

The measurements and the status are retrieved calling the correspondent functions in the AFE and ADC drivers, as well as the state functionalities. Whenever an error occurs, or a problem is detected, the main state of the BMS (which is one of the global variables mentioned before) changes accordingly, and the DSG (or CHG, in case of charging) MOSFET is opened.

Since there's no RTOS included, thus no processes being scheduled, all state transitions and measurements are handled manually and sequentially. The whole loop takes up to 15ms to execute once.

The fact that there are no processes used means that the charging procedure (and thus, the charging state) is triggered by polling the status of a few variables, including the current flowing towards the LVB: if it exceeds a pre-determined threshold, the charging procedure is initiated. This threshold is set to be 1.5A, at the moment. During the *charging procedure*, all measurements (cells voltages and temperatures, battery voltage and current flowing) are taken as usual, with the only difference that they're shown on screen (using the SEGGER RTTOUT routine, which prints messages to the debugger console). The charging procedure is automatically stopped whenever the current flowing to the LVB gets below the threshold, or whenever the LVB voltage exceeds the voltage setpoint of 29.4V, which is the absolute maximum rated voltage for these particular cells.

In both charging and ready state it's possible to balance the cells, given that the LVB is disconnected from the car. This procedure is triggered by a button on the PCB, which unluckily is the same that manually wakes up the MCU from Deep Sleep (due to a mistake in hardware design). This issue has been circumvented using a "de-bouncer", which is just a pure counter that waits before triggering balancing upon pressing the button. After several tests, it has been proven to be the safest solution.

Such counters are used a lot throughout the code, as they allow for more "extended" behaviour of the BMS system. One of these counters is the `CAN update counter`, which allows to send data over the CAN bus at a pre-determined rate (approximately each and every 5 seconds) to prevent overflowing the bus with non-critical data: in our scenario, data coming from the HV system and the sensors have the highest priority on the CAN bus.

2.4.6 LPC-Generated Files

These files are generated automatically when creating a new project using NXP's LPC-based microprocessors, and include a series of routines, functionalities and definitions that help the processor understand where the main entry point of the code is (the "main"), the interrupt handling routines to be called and some startup routines required to correctly initialise the processor to a pre-set state.

It's essential to keep these files in the codebase, since without them the processor wouldn't simply power up correctly.

It's worth mentioning that all the interrupt handler routines consist of a simple empty infinite loop, which blocks the execution of the code and keeps the processor stuck: in order to achieve custom behaviours, it's necessary to re-implement the interrupt handling routines in the code, with the simple:

```
extern "C" __attribute__((interrupt)) void CAN_IRQHandler(void)
{
        /* write your code here */
}
```

If you want to call the pre-defined interrupt handling routing from the system, as it happens in the CAN communication, simply write

```
LPC_CCAN_API->isr();
```

which simply calls the microprocessor's interrupt service routine defined for the CAN peripheral.

Chapter 3

List of Acronyms and Abbreviations

ADC	Analog-to-Digital Converter
AFE	Analog Front End
BMS	Battery Management System
CAN	Controller Area Network
ECU	Electronics Control Unit
GPIO	General-Purpose Input/Output
I2C	Inter-Integrated Circuit
LSB	Least Significant Bit
LVB	Low-Voltage Battery
LVMS	Low-Voltage Master Switch
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MCU	Micro-Controller Unit
PCB	Printed Circuit Board
PLL	Phase-Locked Loop
RTOS	Real-Time Operating System
UART	Universal Asynchronous Receiver-Transmitter

Appendix A

BMS Documents

This appendix contains three important documents to further understand the software contained inside this codebase and this software guide.
In particular, it provides

- The **BMS State Encoder**, as retrieved from the AFE
- The **mapping of the MCU pins**
- The **LED Encoder**, to understand how to recognise each and every state of the LVB+BMS system by simply looking at 7 GPIO-controlled LED's

BMS State Encoder

state_t BMS_STATE

This is an enum containing the possible states of the MCU

state	value (uint8_t)	description
OVERCURRENT	0x01 (1)	Overcurrent detected from the Analog Front End
SHORTCIRCUIT	0x02 (2)	Short circuit in discharge detected from the AFE
OVERVOLTAGE	0x04 (4)	Overvoltage detected from the AFE in one or more cells
UNDERVOLTAGE	0x08 (8)	Undervoltage detected from the AFE in one or more cells
AFE_FAULT	0x20 (32)	AFE is in a fault condition. That can be caused by many factors, for more reference check the data sheet http://www.ti.com/lit/ds/symlink/bq76940.pdf
ERR_TRANSIENT	0x40 (64)	Transient error, can be caused by erroneous readings or glitches. **NEVER SET**
SETUP	0xD0 (208)	BMS is setting up, initialising all pins and peripherals needed to operate, as well as writing some default values to the AFE
SLEEP	0xD1 (209)	BMS is in deep-sleep mode (MCU). All peripherals are disabled, except for the wakeup pins
READY	0xD2 (210)	BMS operational mode. This is the state of the BMS when running and performing every measurement, when clearly there's no error whatsoever
CHARGE	0xD3 (211)	The battery is charging, therefore the BMS regulates balancing operations
OVERTEMPERATURE	0xFE (254)	Overtemperature detected from the MCU from one or more sensors
UNDERTEMPERATURE	0xFF (255)	Undertemperature detected from the MCU from one or more sensors

Important:

- OT and UT are readout directly from the MCU using the thermistor sensors and the ADC converter
- OC, SC, OV and UV are detected by the AFE and then passed to the MCU by reading out the status (*sys_stat*) register.

sys_stat register values

The status register of the AFE contains all the possible error syndromes detected by the circuit, which are then passed to the MCU and influence the bms_state

value	error(s)	description
1	OCD	Overcurrent detected
2	SCD	Short circuit in discharge detected
4	OV	Overvoltage detected
8	UV	Undervoltage detected
16	OVRD_ALERT	External override of the ALERT pin, needs to be cleared to continue working
32	DEVICE_XREADY	AFE chip fault indicator

Clearly, such values can be combined to obtain multiple error conditions at the same time.

Example: error **20** means that there is OVRD_ALERT and OV bits set at the same time, thus two different error syndromes appear simultaneously.

BMS_PinsMapping

Board: LPC11C12FBD48/301

Pins table

Function	Board	Dev Kit (is it soldered?)	Requirement
LED0	PIO2_7	P2_7 (**N**)	output
LED1	PIO2_8	P2_8 (**N**)	output
LED2	PIO2_0/DTR/SSEL1	P2_0	output
LED3	PIO2_6	P2_6 (**N**)	output
LED4	PIO0_3	P0_3	output
LED5	PIO2_5	P2_5 (***not in devkit***)	output
LED6	PIO0_7/CTS	P0_7	output
NRST	RESET/PIO0_0	P0_0 connected to JTAG_RESETX (R32)	input
Alert	PIO0_2/SSEL0/CT16B0_CAP0	P0_2	input
SCL	PIO0_4/SCL	P0_4	output (open-drain)
SDA	PIO0_5/SDA	P0_5	output (open-drain)
CAN_EN	PIO2_4	CAN_L	output
CAN_RX	CAN_RXD	CAN_H	input
CAN_TX	CAN_TXD	Vcc	output
Push	PIO0_6/SCK0	P0_6	input
UART_EN	PIO3_3/RI	P3_3	output
UART_RX	PIO1_6/RXD/CT32B0_MAT0	P1_6	
UART_TX	PIO1_7/TXD/CT32B0_MAT1	P1_7	
WAKEUP	PIO1_4/AD5/CT32B1_MAT3/WAKEUP	P1_4 (**N**)	input **NOT USED**

SWDCLK	SWCLK/PIO0_10/SCK0/CT16B0_MAT2	P0_10 connected to JTAG_TCLK_SWCLKX (R35)	
SWDIO	SWDIO/PIO1_3/AD4/CT32B1_MAT2	P1_3 connected to JTAG_TMS_SWDIOX (R36)	
Temp_EN	PIO3_0/DTR	P3_0 (**N**)	output
Temp0	R/PIO1_1/AD2/CT32B1_MAT0	P1_1	input ADC
Temp1	R/PIO1_2/AD3/CT32B1_MAT1	P1_2	input ADC
Temp2	PIO1_11/AD7	P1_11	input ADC
Sense-	R/PIO1_0/AD1/CT32B1_CAP0	P1_0	input
Sense+	R/PIO0_11/AD0/CT32B0_MAT3	P0_11	input
SenseNeg	PIO0_9/MOSI0/ CT16B0_MAT1	P0_9	input
SensePos	PIO0_8/MISO0/ CT16B0_MAT0	P0_8	input

LED DEFINITION	OK_LED	ERROR_LED	OV_ERROR	OT_ERROR	OC_ERROR	UV_ERROR	UT_ERROR
STATE	G	R	R	R	R	B	B
SETUP							
SLEEP							
READY	X						
CHARGE	X						X
BALANCING	X					X	
CHARGE_AND_BAL	X					X	X
OVERCURRENT					X		
SHORTCIRCUIT					X		
OVERVOLTAGE			X				
UNDERVOLTAGE						X	
AFE_FAULT		X					
I2C_FAIL	X	X	X				
ERR_TRANSIENT		X					X
INVALID_SAMPLE		X	X			X	
ILLEGAL_STATE							
OVERTEMPERATURE				X			
UNDERTEMPERATURE							X

Figure A.1: LED Encoder