Basic Rocket System v1.0

General Requirements and Specification

Rev 01

2025

Lucas Arturo Noce
lucasnoce0@gmail.com
https://github.com/lucasnoce/BRS_v1

Table of Contents

List of Figures

List of Tables

1. Document overview

1.1. Introduction

The Basic Rocket System (BRS) is an open source device designed - as a side personal project - to implement all the most basic functionalities expected from a model rocket avionics computer.

The focus of the project is on firmware development, as hardware is heavily dependent on the rocket structure and design. However, firmware does not exist without hardware, so a block diagram and electrical schematic will be sketched.

The system will be designed to feature some basic functionalities and also to be compatible with future software and, especially, hardware upgrades that can provide further utilities for the system.

1.2. Scope

This document aims to clearly define all the BRS requirements, such as hardware and firmware structure, and also describe guidelines of how the system should perform and be built.

In this document, some main hardware components will also be specified to allow firmware development to be started. The complete electrical schematic and PCB layout are topics for another document.

1.3. Disclaimer

This project is designed by me, Lucas, as a personal project. It is not intended for commercial purposes nor any other use rather than model rocketry hobby activities.

This project is open-source under the MIT license, allowing the community to explore and contribute. However, I disclaim any responsibility for any misuse, damage, or consequences resulting from the use of this project. Do it at your own risk.

Rockets can be seriously dangerous - and potentially deadly - for the ones involved, for others and nearby strangers, so be very cautious, conservative and safe when exploring the rocketry world.

2. Hardware Requirements

2.1. MoSCoW Analysis

The definition of HW specification started by listing all the components related to a standard avionics system. Then, the MoSCoW prioritisation method was implemented in order to enumerate the necessary and desired components. The criteria used to categorize each component was completely arbitrary, but focused on maintaining the lowest possible cost while also allowing the device to have all the key features needed for a basic avionics system.

Table 1 - Hardware Components MoSCoW Analysis

Hardware Component	**MoSCoW**	**Comments**
MCU	Must Have	Focus on low power and/or low cost
IMU	Must Have	Periodically sample acceleration and gyro
Barometer	Must Have	Apogee detection
Power Tree	Must Have	Includes power source, probably LiPo batteries
Recovery System	Must Have	For triggering parachute deployment
Data logging	Must Have	Primarily flash chip, maybe microSD card for cost reduction
Remove Before Flight	Must Have	Must be removed with the rocket fully assembled
LEDs	Must Have	Indicate system state
Basic Backup System	Should Have	May not be designed for cost reduction
Buzzer	Should Have	Indicate system state
GPS	Should Have	May not be used for cost reduction
Wireless Communication	Should Have	Will likely be added later, probably LoRa P2P
Camera	Could Have	Really increases system complexity and storage demand, may be added later as a separate system
Self Ignition System	Could Have	Depends on rocket motor design

Because of the mentioned criteria, some important components were categorized as “Should Have” or even “Could Have” to reduce costs. But, as the project evolves, some of them may be added on later versions of the system.

After the MoSCoW analysis, the components chosen to integrate the project are the following:

- MCU
- IMU

- Barometer
- Power Tree
- Recovery System
- Data logging
- Remove Before Flight
- LEDs
- Buzzer

2.2. Components Specification

Considering that the device will not be mass produced, many components were chosen based on availability, meaning that the focus on low power and/or low cost are often more dependent on retailers stock than on datasheet information.

2.2.1. MCU

For the MCU, the following options were considered. Component information was sourced from the corresponding datasheet. Average price is an approximate value taken from local stores.

Table 2 - MCU Options

Manufacturer	**Part #**	**Flash / RAM (bytes)**	**Max Clock (MHz)**	**Consumption (mA@$freq_{max}$)**	**Average Price**
STMicroelectronics	STM32F103C8	64k / 20k	72	50	R$ 30,00
STMicroelectronics	STM32F103RB	128k / 20k	72	50	R$ 100,00
STMicroelectronics	STM32F401CC	256k / 64k	84	22	R$ 45,00
STMicroelectronics	STM32F411CE	512k / 128k	100	23.6	R$ 55,00
Espressif	ESP32-WROOM-32	4M / 520k	240	80	R$ 40,00

Data driven analysis indicates that the best option (low cost, high density) would be the ESP32. However, previous experiences were also considered and I happen to not be a huge fan of the Espressif platform. Besides, the datasheet shows an average current consumption of 80 mA, which is the highest value amongst the options.

The second best component seems to be the STM32F411CE, which has enough flash memory to store data from one entire flight and the largest RAM compared to the remaining options. That, alongside presenting almost a quarter of the current consumption than the ESP32, has made this the selected MCU.

2.2.2. IMU

For the IMU, the following options were considered. Component information was sourced from the corresponding datasheet. Average price is an approximate value taken from local stores.

Table 3 - IMU Options

Manufacturer	Part #	DOF	Max Acc ODR (Hz)	Max Acc Range	Supply Voltage	Consumption	Bus	Average Price
NXP	MMA8452	3	800	±8g	2.0~3.6V	<165 µA	I2C	R$ 20,00
InvenSense	MPU-6050	6	1k	±16g	2.4~3.4V	<3.9 mA	SPI / I2C	R$ 30,00
STM	LSM6DS3	6	1.6k	±16g	1.7~3.6V	<900 µA	SPI / I2C	R$ 20,00
Bosch	BNO055	9	100	±16g	2.4~3.6V	<12.3mA	I2C / UART	R$ 300,00

// TODO

2.2.3. Barometer

// TODO

2.2.4. Power Tree

The Power Tree (PWT) is actually a subcircuit that contains all power supply components, as well as the control signals and other power related items. As of the first version of this system, the PWT block diagram is as follows. The red arrows indicate power delivery traces and the grey ones indicate signal traces.

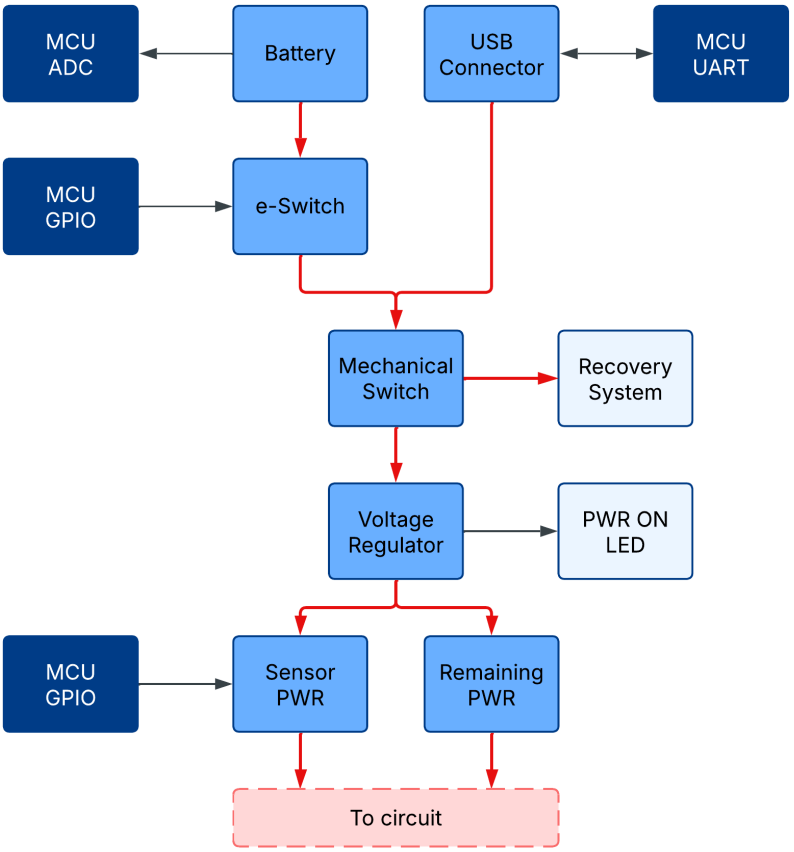

Figure 1 - Power Tree block Diagram

The subcircuit provides two ways of powering the device: internal battery and USB connector. To prevent issues with both power sources connected at the same time, protective diodes must be added accordingly and an SMD solder jumper should be added to allow physical disconnection of the USB power line to the system, still granting serial communication through GND and UART traces.

The internal battery must be able to provide the voltage and current necessary to the system through the whole flight process, including the periods between system power up and the actual vehicle launch and between landing and physical recovery of the rocket. Both of these periods are unknown beforehand and prone to several minutes or hours of delay, depending on a variety of factors such as weather conditions, unexpected on-site issues, etc.

With that in mind, it is important to calculate, simulate and test a large number of scenarios in order to correctly specify the minimum battery capacity needed, which will only be possible to do reliably after the whole system is working properly. Thus, the current capacity of the internal battery will be defined later on.

The voltage however, depends on two main factors: the voltage regulator constraints and the recovery system design. The first factor will probably not be an

issue, as the voltage regulator selected will most certainly be a switching regulator, which typically accepts a wide range of input voltage.

The recovery system design can vary a lot. In my previous experience, this system was quite unusual and deployed the parachutes from a side door on the body of the rocket without the use of any explosive. Instead, it used some electromagnets to hold a metal piece fixated on the side door. This design is interesting for not using explosives, but continuously drains current from the power supply to keep the doors closed and also requires higher voltages to properly activate the electromagnets.

A more common approach is to divide the rocket into two (or more) parts and position the parachutes on the nose cone. Some sort of explosive is then used to separate these parts and deploy the recovery system. In this method, despite having some associated risk related to the explosive, the power demand is significantly smaller, since the system only uses power momentarily to ignite the explosive.

There are plenty of other designs and philosophies regarding recovery systems. To simplify things a little, this project will use a 12V battery source, which is a fairly common value and may be compatible with most recovery system designs.

Back to the PWT subcircuit, some details are important to guarantee a successful flight and to improve avionics system handling on a fully assembled rocket. For that, the PWT design shall follow the guidelines:

- ON/OFF mechanical switch must be accessible from outside with the rocket fully assembled;
- PWT subcircuit shall not feature battery charging for the sake of simplicity;
- USB connector shall be either type C (preferably) or microB;
- A solder jumper must be added to select between power sources, i.e. a jumper footprint on the USB power line right before the Mechanical Switch, effectively allowing the disconnection of the USB power supply;
- The system must implement a self-shutdown feature controlled by MCU, through the e-Switch;
- The voltage regulator must be able to provide enough power for all transient events that are known;
- It is advisable that the implemented regulator follows a switching topology for better overall performance;
- The MCU must be able to control the power delivery (ON/OFF) to the “sensors” (IMU, BAR and backup flash);

2.2.5. *Recovery System*

// TODO

2.2.6. *Data Logging*

// TODO

- Flash chip
- Rough data structure (24 bytes):
    - (4B) Timestamp
    - (6B) Acc [x,y,z]
    - (6B) Gyro [x,y,z]
    - (4B) Pressure
    - (2B) Temperature
    - (2B) Battery Voltage
- Absolute minimum data log flash size: 256 kB
- Variable sample/store rate based on state

// TODO: OpMode Selector switch

2.2.7. *Remove Before Flight*

// TODO

- Must be accessible from outside, i.e. with the rocket fully assembled
- Must be possible to reinsert the tag with the rocket fully assembled

2.2.8. *LEDs and Buzzers*

To define the number of LEDs necessary to the system, the following reasoning was presented:

- FW predicts a total of 11 states on State Machine, so a 4 LEDs can display the current state (for debugging only, they may be removed for flight)
- 1 LED for power input indication (purely HW, may be removed for flight)
- 1 LED for low battery indication
- At least 2 more LEDs for other indications and debugging

With this in mind, a total of 8 LEDs can comfortably handle all the needs of the system. Since one of them is not controlled by the MCU, there are 7 LEDs left to

be controlled. Using one dedicated GPIO for each LED is poor use of resources and can lead to unnecessary future constraints on the hardware side, so a better option is to control the LEDs through an I2C IO Expander. And because these ICs typically provide a power-of-two number of IOs, one more LED will be added to complete 8 controlled LEDs and 1 uncontrolled LED.

For the expander, some options were considered and the selected IC was the TCA9534A by Texas Instruments, since it provides all the needs for this task as well as being the cheapest option.

The buzzer, on the other hand, can't be efficiently used through an IO Expander, since it requires PWM output for better control. Hence, it will be connected to a dedicated PWM-capable GPIO. Also, only one buzzer will be used, since there are only 3 use cases predicted, which don't interfere with each other:

- Signaling RBF
- Facilitate recovery process (keeps beeping after landing)
- Low battery indication

Finally, it is worth noting the suggestion to add a low pass filter between the PWM GPIO and the buzzer input to increase sound quality.

2.2.9. Components Summary

Finally, a summary of all selected components and corresponding specification is presented below. This list is still not a Bill Of Materials, so only the most important components are shown.

Table 4 - Components Summary

Component	**Specification**	**Average Price**
MCU	STM32F411CE	
IMU		
BAR		
Power Tree	x V battery, USB, ON/OFF switches	
Recovery		
Backup Flash		
RBF		
LEDs	8 LEDs through TCA9534A IO expander	
Buzzer	1 buzzer	

2.3. Block Diagram

Based on the selected components, a high abstraction block diagram was designed to help visualize the circuits. Grey arrows indicate signal traces, while the green ones indicate high voltage and red ones low voltage power delivery traces.

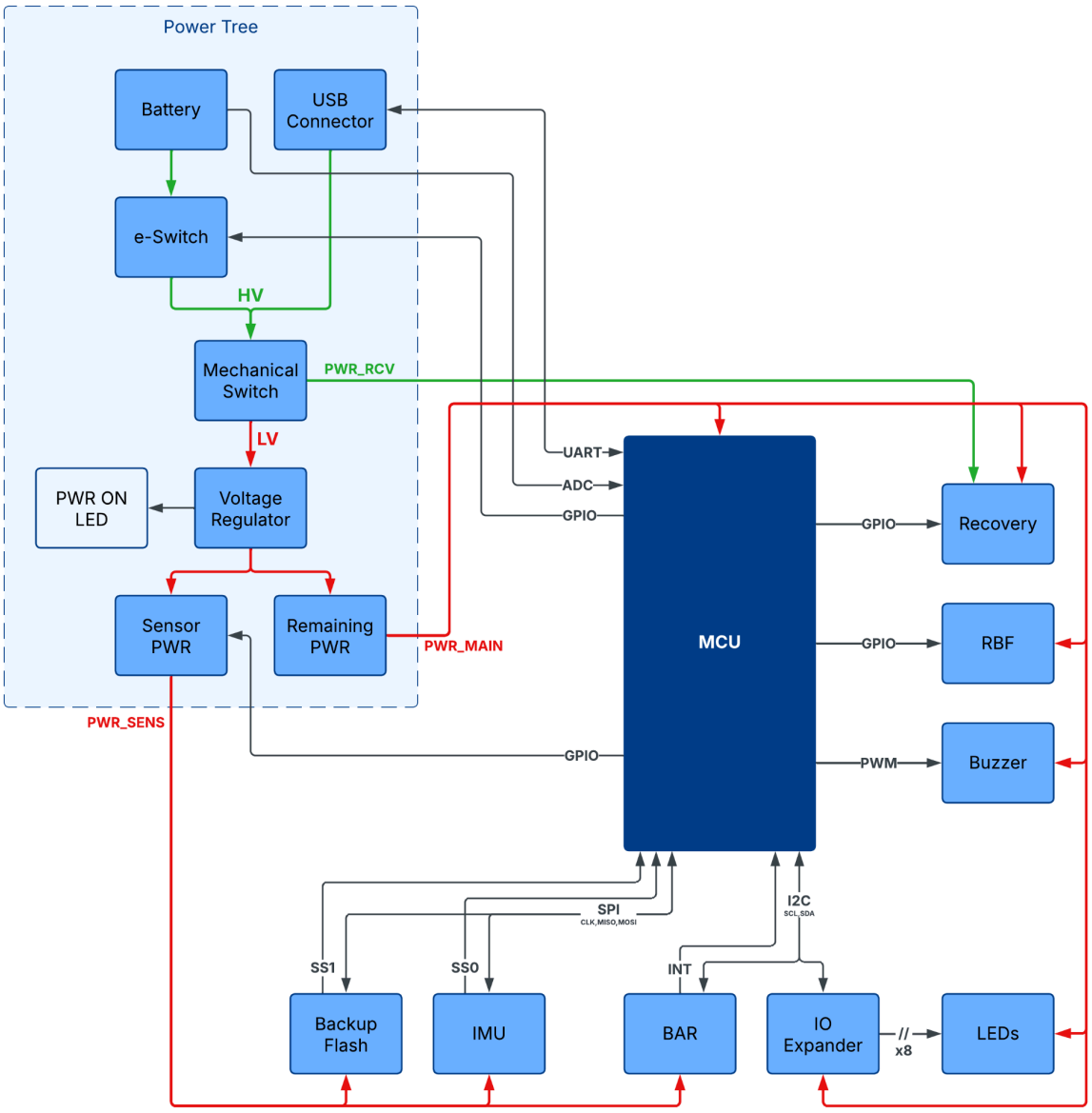

Figure 2 - Circuit Block Diagram

3. Software Requirements

3.1. State Machine

A state machine was created to better organize and control the device under all the possible flight stages and implement other functionalities.

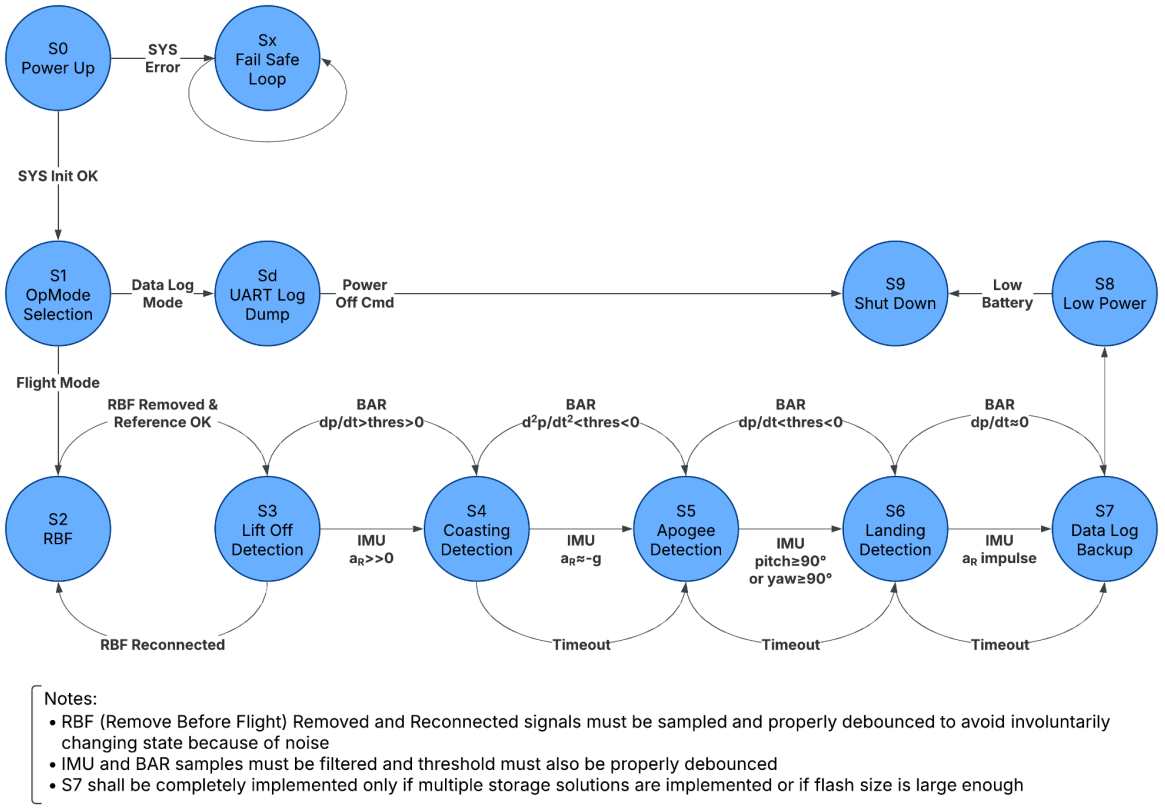

Figure 3 - System State Machine

3.1.1. S0

This is the first state of the system after being powered up. It carries the responsibility of initializing all the drivers and services necessary for device operation.

Since the system can wake up after a variety of events - and, therefore, have valuable data stored in flash memory - the S0 state is also responsible for reading some system status information from flash and deciding how to move forward with the operation from a safety perspective. It should be able to detect, for example, if the system is booting up mid flight after some unexpected shutdown has happened and after that, try to resume operation from the previously known state.

3.1.2. Sx

This is a fail safe state, designed to safely and reliably manage system initialization errors, allowing unexpected behavior to be handled in a controlled and harmless way.

3.1.3. S1

This state is responsible for determining the selected operation mode based on a physical switch/jumper. There are 2 operation modes:

- Flight mode corresponds to a secondary state machine which tracks the flight stages based on sensor readings. It consists of states S2 through S7.
- Data Logger mode corresponds to a read-only state where stored flight data can be safely accessed and retrieved from internal and/or external storage.

// TODO: setup and test mode may be implemented later.

3.1.4. S2

This state is the first step of the flight mode. It continuously tries to identify if the Remove Before Flight (RBF) tag has been removed, indicating that the vehicle is completely assembled and ready to launch. This is done by polling the corresponding RBF GPIO input value and getting the RBF removed value for many times in a row.

Also in this state, right after the RBF removal is detected, some reference measures are taken and used as a starting point for the flight, such as vehicle attitude (inclination of the rocket), vehicle orientation (the axes with the largest acceleration magnitude), ground level pressure, etc. These measures must be successfully completed in order for the machine to advance to the next state, otherwise it must indicate this error and stop the launch procedure for vehicle inspection and system reboot.

3.1.5. S3

After S2 is completed, the machine enters a lift off detection state. In this state, the system effectively starts to read, interpret and store data from the sensors continuously with the objective of detecting the motor ignition. These are the possible signals that can trigger a state change (in order of priority):

- RBF reconnected: if the RBF is reconnected, the system will understand it as a launch hold and return to the previous state (S2).

- IMU resulting acceleration values show a sudden and intense increase in magnitude while also inverting the signal value of the vertical orientation axes (previously determined).
- BAR readings show a sudden, intense and consistent decrease in magnitude, indicating a sequential drop in pressure, thus a rise in altitude.

3.1.6. S4

After successfully detecting lift off, the system moves to S4 which aims to detect the ending of powered ascent and beginning of coasting phase. For this to happen, these triggers are considered (in order of priority):

- IMU resulting acceleration values show a consistent decrease in magnitude (tending to the previously determined baseline value of approximately -1g) while also inverting the signal value of the vertical orientation axes again.
- BAR readings show a growing decrease in magnitude (approximately constant decrease of second derivative), indicating a sequential drop in pressure, thus a rise in altitude
- A coasting detection timeout occurs (empirically determined and implemented in pre-compile time).

3.1.7. S5

After successfully detecting coasting, the system moves to S5 which aims to detect apogee, i.e. the ending of ascent and beginning of descent phase. For this to happen, these triggers are considered (in order of priority):

- BAR readings show a consistent increase in magnitude, indicating a sequential rise in pressure, thus a drop in altitude. Can also be understood as a change in slope signal of the pressure variation.
- IMU gyroscope values show an absolute rotation above 90° in pitch (θ) or yaw (φ), not considering rotation around the previously determined vertical orientation axes, roll (ζ).
- An apogee detection timeout occurs (empirically determined and implemented in pre-compile time).

3.1.8. S6

After successfully detecting apogee, the system moves to S6 which aims to detect landing. For this to happen, these triggers are considered (in order of

priority):

- BAR readings show consistently similar magnitude values, indicating steady pressure levels, thus a stable and constant altitude.
- IMU resulting acceleration values show a short impulse magnitude (full scale) followed by constant values of around 1g.
- A landing detection timeout occurs (empirically determined and implemented in pre-compile time).

3.1.9. S7

After landing detection, the system enters Data Log Backup state, which can be skipped if there are no backup storage devices detected. This state copies the contents of the internal storage into the backup storage for redundancy and ease of access, especially if the backup device is an SD card.

3.1.10. S8

After backing up the stored flight data, the system goes to Low Power state, where it deinitializes all the unnecessary drivers and services and enters a light sleep loop. In this loop, the device keeps blinking an indication LED and beeping the buzzer for some period to help find the vehicle on the recovery mission.

This loop continues until a low voltage threshold is reached on the ADC readings of the internal battery. After that, the machine moves to the S10 state.

3.1.11. S9

In this state, the system executes a self-shutdown procedure. This procedure triggers a MOSFET to essentially disconnect the internal battery (main power source during flight) from the power supply line, which turns the device completely off.

3.1.12. Sd

This state corresponds to UART Log Dump, which allows the device to export the flight data stored in either the internal or backup storage. The device waits for a read command and, once one is received through a serial port (USB or UART), it loops through the whole storage device printing the values in a standardized format on both serial ports.

3.2. Flowchart

Based on the state machine, the following firmware flowchart was designed. WARNING: out of date.

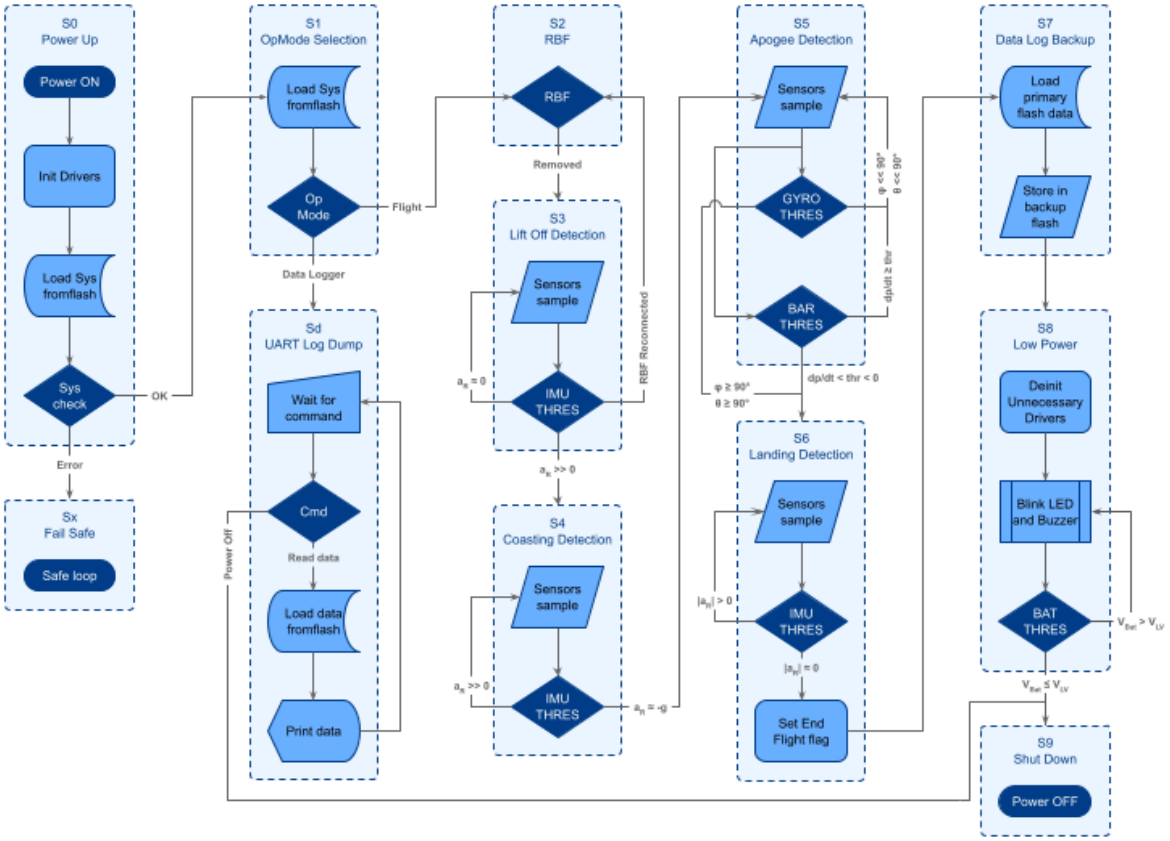

Figure 4 - Firmware Flowchart

3.3. Firmware Architecture

The firmware abstraction layers are organized as illustrated below. The MCU HAL layer is provided by STM32CubeMX and allows the use of MCU hardware. Above the MCU HAL, a Board Support Package provides an isolation layer between the BRS and hardware specific functions, essentially granting a high level of compatibility between the whole software above and almost any other MCU with minimal driver refactoring.

The Drivers layer implements all the necessary drivers for the suggested hardware approach mentioned in the previous section. This layer is then consumed by the Services layer, which manages all the necessary activities needed. Finally, a high level Application layer implements the designed state machine. Between the Services and Application layers is the Inter Process Communication service which

provides a messaging system for threads of both layers to communicate through. The RTOS used is still to be defined, but will most likely be FreeRTOS.

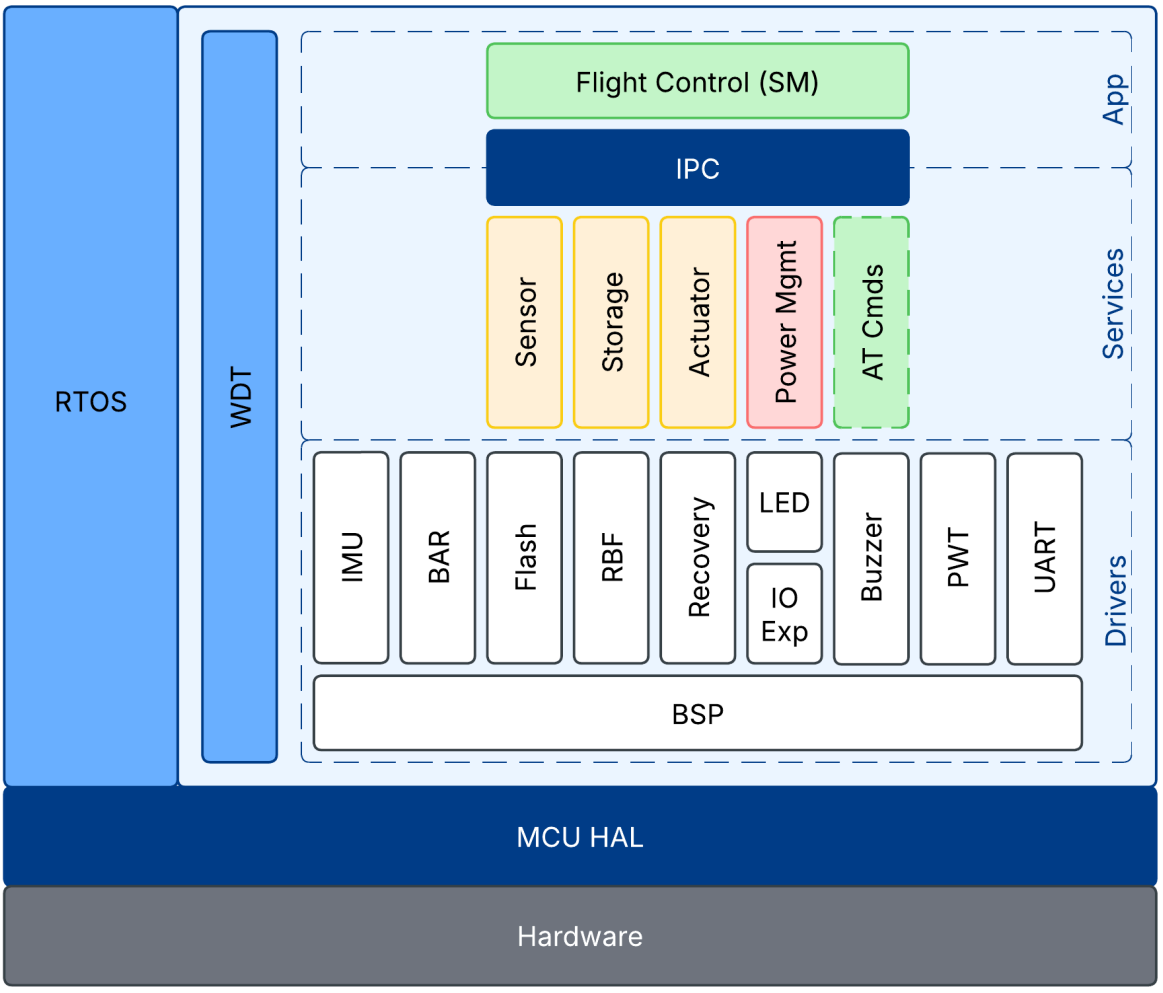

Figure 5 - Firmware Layer Architecture